NAVAL POSTGRADUATE SCHOOL

Monterey, California

THESIS

A SPLIT-AND-MERGE METHOD FOR CREATING POLYGONAL HOMOGENEOUS-VEGETATION REGIONS FROM DIGITIZED TERRAIN DATA

by

Roderick K. Wade

June 1989

Thesis Advisor: Neil C. Rowe

Approved for public release; distribution unlimited

Unclassified

Security Classification of this page

REPORT DOCUMENTATION PAGE

1a Report Security Classification: UNCLASSIFIED	1b Restrictive Markings
2a Security Classification Authority	3 Distribution Availability of Report: Approved for public release; distribution is unlimited.
2b Declassification/Downgrading Schedule	
4 Performing Organization Report Number(s)	5 Monitoring Organization Report Number(s)
6a Name of Performing Organization: Naval Postgraduate School; 6b Office Symbol (*If Applicable*): 52	7a Name of Monitoring Organization: Naval Postgraduate School
6c Address (*city, state, and ZIP code*): Monterey, CA 93943-5000	7b Address (*city, state, and ZIP code*): Monterey, CA 93943-5000
8a Name of Funding/Sponsoring Organization; 8b Office Symbol (*If Applicable*)	9 Procurement Instrument Identification Number
8c Address (*city, state, and ZIP code*)	10 Source of Funding Numbers: Program Element Number; Project No; Task No; Work Unit Accession No

11 Title (*Include Security Classification*)
A SPLIT-AND-MERGE METHOD FOR CREATING POLYGONAL HOMOGENEOUS-VEGETATION REGIONS FROM DIGITIZED TERRAIN DATA

12 Personal Author(s)
Roderick K. Wade

13a Type of Report	13b Time Covered	14 Date of Report (*year, month,day*)	15 Page Count
Master's Thesis	From To	June 1989	89

16 Supplementary Notation
The views expressed in this thesis are those of the authors and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

17 Cosati Codes			18 Subject Terms (*continue on reverse if necessary and identify by block number*)
Field	Group	Subgroup	Computer Vision, Artificial Intelligence, Optimal-Path-Planning. Digitized Terrain Databases, Prolog

19 Abstract (*continue on reverse if necessary and identify by block number*)

Providing a simplified representation of terrain characteristics has applications to optimal-path-planning programs using spatial reasoning. Utilizing computer vision techniques, our program creates polygonal homogeneous-vegetation regions based on map vegetation data from a digitized Defense Mapping Agency Database. Boundary points for regions are identified from the vegetation codes in the database, and then the boundary contours of the regions are traced using a modified look-left boundary tracing algorithm. Each region is then represented by a polyline comprised of line segments that meet a minimum threshold for fit using the linear least-squares criterion. The segments are determined by first recursively splitting the region boundary until all segments meet the fit threshold, and then merging adjacent segments that meet the threshold.

20 Distribution/Availability of Abstract	21 Abstract Security Classification
[X] unclassified/unlimited [ ] same as report [ ] DTIC users	UNCLASSIFIED

22a Name of Responsible Individual	22b Telephone (*Include Area code*)	22c Office Symbol
Prof. Neil C. Rowe	(408) 646-2462	Code 52Rp

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted security classification of this page

All other editions are obsolete Unclassified

Approved for public release; distribution is unlimited.

A Split-and-Merge Method for Creating Polygonal Homogeneous-Vegetation Regions from Digitized Terrain Data

by

Roderick K. Wade
Captain, United States Army
B.S., United States Military Academy, 1981

Submitted in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 1989

ABSTRACT

Providing a simplified representation of terrain characteristics has applications to optimal-path-planning programs using spatial reasoning. Utilizing computer vision techniques, our program creates polygonal homogeneous vegetation regions based on map vegetation data from a digitized Defense Mapping Agency database. Boundary points for regions are identified from the vegetation codes in the database, and then the boundary contours of the regions are traced using a modified look-left boundary tracing algorithm. Each region is then represented by a polyline comprised of line segments that meet a minimum threshold for fit using the linear least-squares criterion. The segments are determined by first recursively splitting the region boundary until all segments meet the fit threshold, and then merging adjacent segments that meet the threshold.

TABLE OF CONTENTS

I. INTRODUCTION

A. BACKGROUND

Recent work at the Naval Postgraduate School has studied methods to determine the optimal path between two points in terrain. These methods assume that the terrain can be represented by homogeneous polygonal regions. Attributes of a region can include the degree of vegetation, type of vegetation, soil composition, elevation, slope, orientation of the slope, and man-made physical features within the area. Currently the optimal-path-planning research is using artificial terrain data. A program that will take Defense Mapping Agency digitized terrain data and create homogeneous regions would enable researchers to show the real-world feasibility of their path-planning approaches. In this thesis, we have developed tools to create these regions from vegetation data recorded at evenly-spaced sample points.

To create two-dimensional regions from vegetation data at evenly-spaced points we used two phases. The first phase identified the boundary points of a homogeneous region, points halfway between adjacent sampled points of differing vegetation. The second phase used a split-and-merge method with linear-least-squares constraints to reduce the complexity of the region boundaries (their number of vertices). The simplified regions enable us to represent the terrain characteristics in a clear and concise manner.

B. ORGANIZATION

Chapter 2 introduces previous work in the areas of region representation by polylines, split-and-merge algorithms, and the linear-least-squares constraints method of approximating the fit of points to a line. Chapter 3 describes the Defense Mapping Agency Databases, and databases used for optimal path planning. In Chapter 4 we discuss in detail our program. Chapter 5 shows our experimental results in both qualitative and quantitative terms. Finally, Chapter 6 summarizes our contributions and discusses some of the possible areas for further research based on this work.

II. REVIEW OF REGION REPRESENTATION BY POLYGONS

A. POLYLINES

Polylines can be used to approximate the boundary of a region. A polyline representation consists of a list of points. See Figure 1. A region can be represented

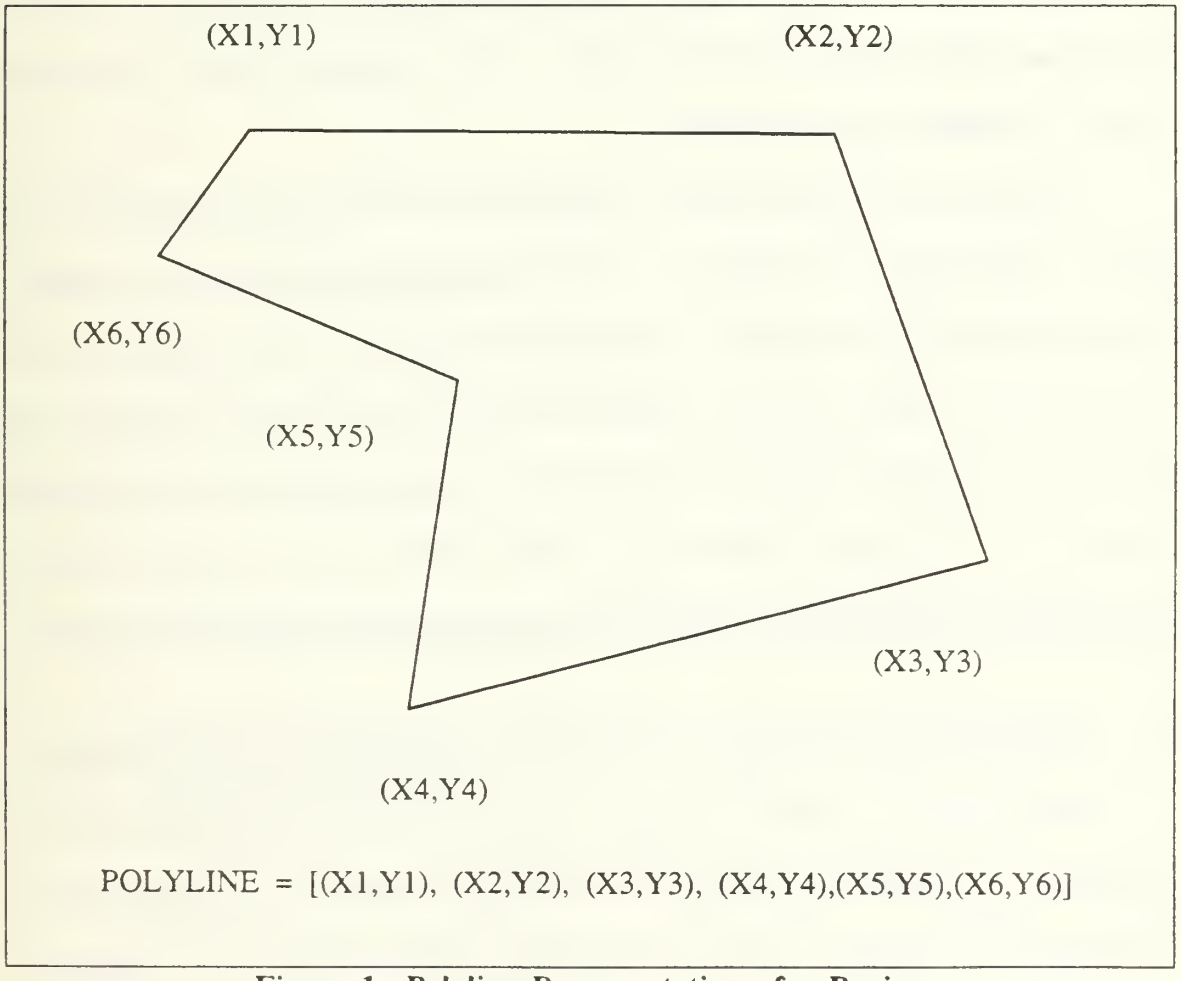


Figure 1 Polyline Representation of a Region

to any degree of accuracy by the polyline depending upon where and how many points are used [Ref. 1:p. 232].

A common technique used to determine a polyline representing a region is splitting and merging. It can be proved that the number of line segments in a polyline constructed using a split and merge algorithm will never be greater than two times the minimum number of segments for a given line fit criteria [Ref. 2: p.283]. There are numerous methods to determine whether a given line segment should be split, merged or left untouched. Usually all methods split segments of a polyline until all subsegments meet criteria for linearity. Then usually adjacent segments of the polyline are merged together if they meet other criteria. The methodology used to determine linearity is problem-domain-dependent.

The linear least-squares method to determine the collinearity of points has been used in a merge-only scheme [Ref. 3]. Two adjacent points were arbitrarily selected and points added to their segment until the linear least-square fit of the segment failed the fit criteria. At that point a new segment was started and the process repeated until the region was completed. The start point for each segment determines the breakpoints of the region. The primary disadvantage of this method is that the merging without backtracking or splitting does not give the most aesthetically pleasing breakpoints.

B. APPROXIMATION OF FIT BY LINEAR LEAST-SQUARES METHOD

The fit of a set of points with respect to a given line can be judged using several techniques. The linear least-squares method sums the squares of the distances from each data point to the line. In our work the line is determined by the two endpoints of the boundary-points subset tested. We use the general equation of a line:

$$Ax + By + C = 0 \qquad (1)$$

Given two endpoints (X1,Y1) and (X2,Y2) of the line segment we can derive the value of the constants A, B, and C by:

$$\frac{Y - Y1}{X - X1} = \frac{Y2 - Y1}{X2 - X1} \qquad (2)$$

Cross multiplying and solving for zero yields:

$$A = Y1 - Y2 \qquad (3)$$

$$B = X2 - X1 \qquad (4)$$

$$C = X1Y2 - X2Y1 \qquad (5)$$

The slope and the intercept are:

$$M = -A / B \qquad (6)$$

$$B = -C / B \qquad (7)$$

The linear least-squares fit for n points with respect to the line calculated above is:

$$FIT = \sqrt{\frac{\sum_{i=1}^{n} (Ax_i + By_i + C)^2}{n(A^2 + B^2)}} \qquad (8)$$

The fit is a calculation of how far on the average the n points are from the line. A fit of zero means that all of the points lie on the line. By squaring the distance from the line we give equal weight to points that are on each side of the line. The square root enables us to renormalize distance from the line.

III. SPATIAL TERRAIN DATABASES

A. DEFENSE MAPPING AGENCY DATABASES

The Defense Mapping Agency maintains digitized terrain databases for most of the world. These databases contain information about elevation data, vegetation, bodies of water, and man-made objects. Typically these databases record data at evenly spaced sample points in latitude and longitude.

The Digital Terrain and Elevation Data (DTED) database used in our program contains two types of such information. The terrain elevation and the height of the vegetation coverage are encoded in two bytes for each sample point. The three most significant bits are the vegetation code and the remaining 13 bits are the elevation in feet. The three bits of vegetation code are explained in Figure 2. Vegetation codes 6 and 7 never occured in the terrain samples we selected so we did not provide additional handling for these codes. The samples are conducted every 12.5 meters. Each square kilometer or grid square requires 6400 samples (80 * 80) or 12,600 bytes.

B. DATABASE USAGE FOR OPTIMAL-PATH PLANNING

There are several approaches to determine the optimum path between two points. At the Naval Postgraduate School considerable research is being conducted using spatial reasoning. Spatial reasoning methods do not use traditional grid terrain modeling where the terrain is represented by evenly distributed sample points. Instead spatial reasoning uses descriptive terrain modeling where the terrain is partitioned into

VEGETATION CODE	VEGETATION HEIGHT
0	LESS THAN 1 METER
1	1 - 4 METERS
2	4 - 8 METERS
3	8 - 12 METERS
4	12 - 20 METERS
5	GREATER THAN 20 METERS
6	NO DATA AVAILABLE
7	NOT USED

Figure 2 Vegetation Codes

homogeneous regions. The minimal-energy optimal-path-planning research conducted by Ron Ross uses partitioned regions based on the slope of the terrain, soil composition, and other factors [Ref 4].

The first step to implement this work was done by Seung Hee Yee who wrote a program for planar-patch terrain modeling based on the elevation data [Ref 5]. One of the three methods that he tested was joint top-down and bottom-up terrain modeling. His top-down phase used a quadtree subdivision method to divide the terrain area into regions represented by a plane and the fit of that plane to the data points. After all subregions meet the appropriate fit threshold, he used a bottom-up approach to merge similar adjacent regions. His bottom-up phase used two merging criteria. First, the adjacent planes must have a minimum difference of the squares of

the differences of the respective plane coefficients. Second, the data points from both regions must be representable by a plane which meets the same fit threshold as in the top-down phase. If both criteria are met, then the regions are merged and the new plane is stored along with its fit. During the year since Seung Hee Yee developed his program, Professor Rowe has improved it by including better techniques to insure continuity between the planes of adjacent regions. The planar patches or regions from these programs could be further partitioned into regions of homogeneous vegetation, but this has not yet been done.

The simplest minimal-energy optimal-path-planning programs run on the order of N squared with respect to the number of vertices per region and the number of regions. For this reason it is important to create regions which preserve the topology of the original sampled data and yet minimize the number of vertices.

IV. IMPLEMENTATION

The program developed is a method to create polygonal homogeneous-vegetation regions from the Defense Mapping Agency database described in Chapter 3. To form these regions the program's first phase identifies which sample points in the database have different vegetation codes than their neighbors. These sample points are identified as boundary cells. A boundary-identification algorithm is then used on these boundary cells to determine sequences of boundary points between regions of different vegetation. A second phase then takes the sequence of boundary points for each region and performs a split-and-merge algorithm on the list of those points to redefine the regions in a more efficient form (one with fewer vertices).

A. DESCRIPTION OF THE BOUNDARY-IDENTIFICATION PHASE

We chose C as the language for the boundary-identification phase. The program runs using a BSD 4.3 compiler on a VAX 11/785 or on an Integrated Solutions (ISI) workstation. Appendix B contains the source code for the boundary indentification phase. This phase requires input from the Defense Mapping Agency DTED database. This database consists of elevation and vegetation data points every 12.5 meters. Our program builds regions containing sample points with the same vegetation code for a one kilometer by one kilometer grid square.

Several assumptions were required in the boundary-identification phase. First, each region formed is assumed to contain vegetation of only one kind. This is

fundamental to the concept of homogeneous regions. Second, regions will not consist of one sample point. These singular points which have no adjacent neighbors above, below, or beside them are changed to have the same code as the adjacent data point with the largest vegetation code (i.e. heaviest vegetation). Third, holes in regions will be detected only in the sense that another region will be formed inside of the outer region. No special classification is added to a region if it has a hole.

The boundary-identification phase is divided into two passes through the input data. The first pass identifies the boundary cells. A sample point is considered to be a boundary cell if any of its four neighbors above, below, or beside it have a different vegetation code. The sole exception to this rule exists when the point itself or its differing neighbor are singular points, which are not considered to be boundary cells in accordance with the second assumption. An example of vegetation codes and identified boundary cells is shown in Figure 3.

The second pass searches through the identified boundary cells and traces the contour for each region; as boundary cells are traced, they are marked and identified with a specific region number. Region numbers are generated as necessary beginning with 1. The search for a boundary cell to start the contour-tracing for each region is started in the upper left-hand corner of the input. The grid is searched in a left-to-right raster scan stopping at the first boundary cell that has not been assigned to a specific region. After this region has been traced by the contour-tracing algorithm from this start point, the raster scan continues.

The contour-tracing algorithm follows the traditional eight-connected look-left algorithm for traversing a maze [Ref. 6:p. 278]. At each boundary cell during the trace

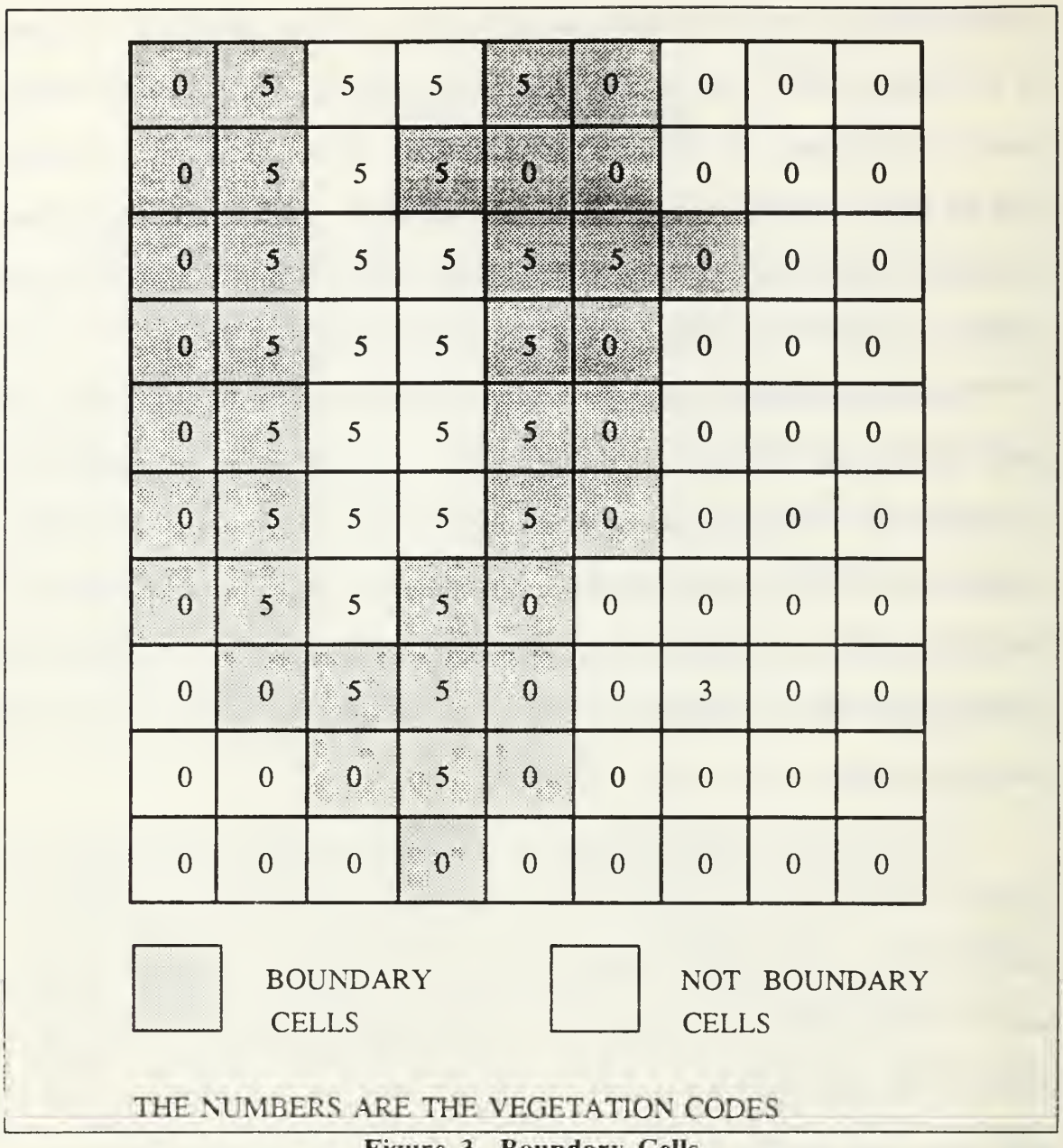


Figure 3 Boundary Cells

the direction of entry determines which way we look to find the next boundary cell; the direction of entry at the start point is defined to be the raster scan direction. Normally, we look to the boundary cell to the left of the direction of entry. If this cell is not a boundary cell of the same vegetation code we look one cell left and

forward. This process is continued clockwise until we find a boundary cell. The contour-tracing algorithm is continued until we reach the start point or a border of the picture (the first pass ensures that either event must eventually occur).

Several additions to the contour-tracing algorithm were included to enable us to handle conditions along the exterior borders of the overall 1km. by 1km. area. When the contour tracing reaches an exterior border, a check is made to determine if a border has previously been encountered for the current region. If a border has not previously been encountered, the tracing continues in the opposite direction starting at the initial point; otherwise, tracing stops. See Figure 4. Second, if the initial point of the region is next to the exterior border the tracing will start away from the border. See Figure 5.

A significant improvement in the representation of the data during this pass is gained by treating the actual boundary between two regions as the set of all points half-way between the centers of each pair of adjoining boundary cells with a different vegetation code. Tracing thus proceeds between these points. This is similar to the crack edges approach used to represent the boundary between regions in [Ref. 1:p. 78]. Additionally, this technique smooths the staircase effect created by the crack edges approach. See Figure 6.

The boundary-tracing phase trades memory space for clarity of code. The input is stored in an 80 x 80 array which represents the vegetation codes at each of the 6400 sampled points in row-major order. The first pass places boundary cells in a 6400 x 5 array in row-major order, classifying each by its x-coordinate, y-coordinate, vegetation code, whether it is a boundary point, and a flag for marking when the

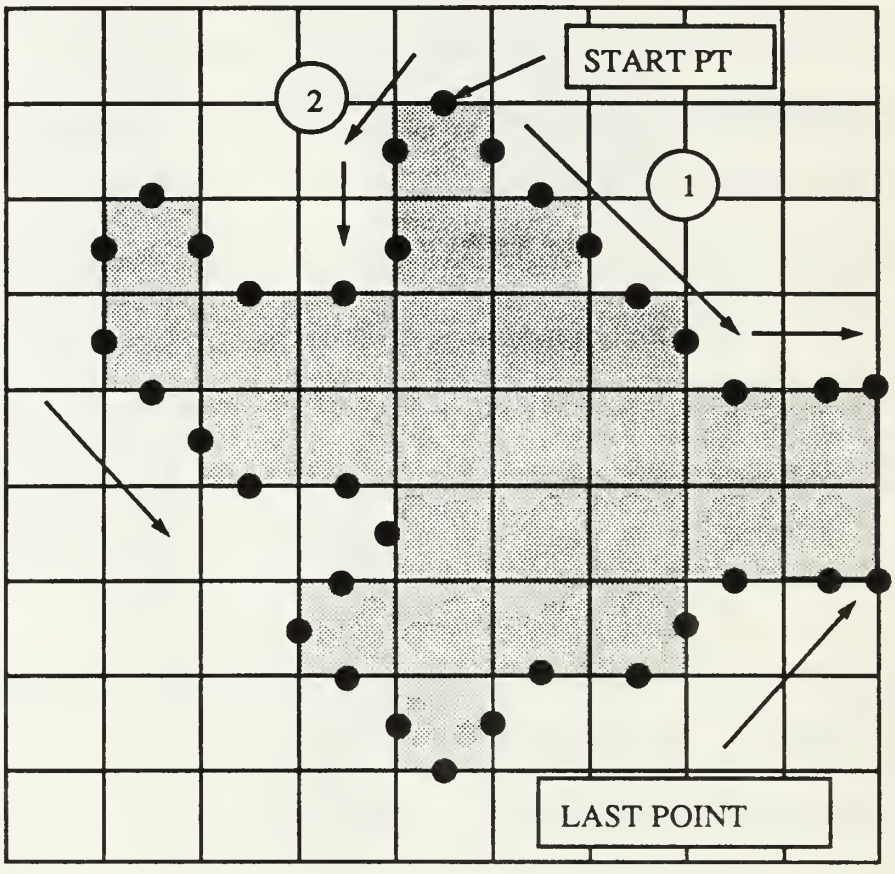


Figure 4 Tracing after Hitting the Border

second pass places the boundary point in a specific region. The second pass stores into a separate array for each region the sequenced boundary points x-coordinate, y-coordinate, and vegetation code. The array of regions is output to a formatted file which will be used as input to the split-and-merge phase described in section B.

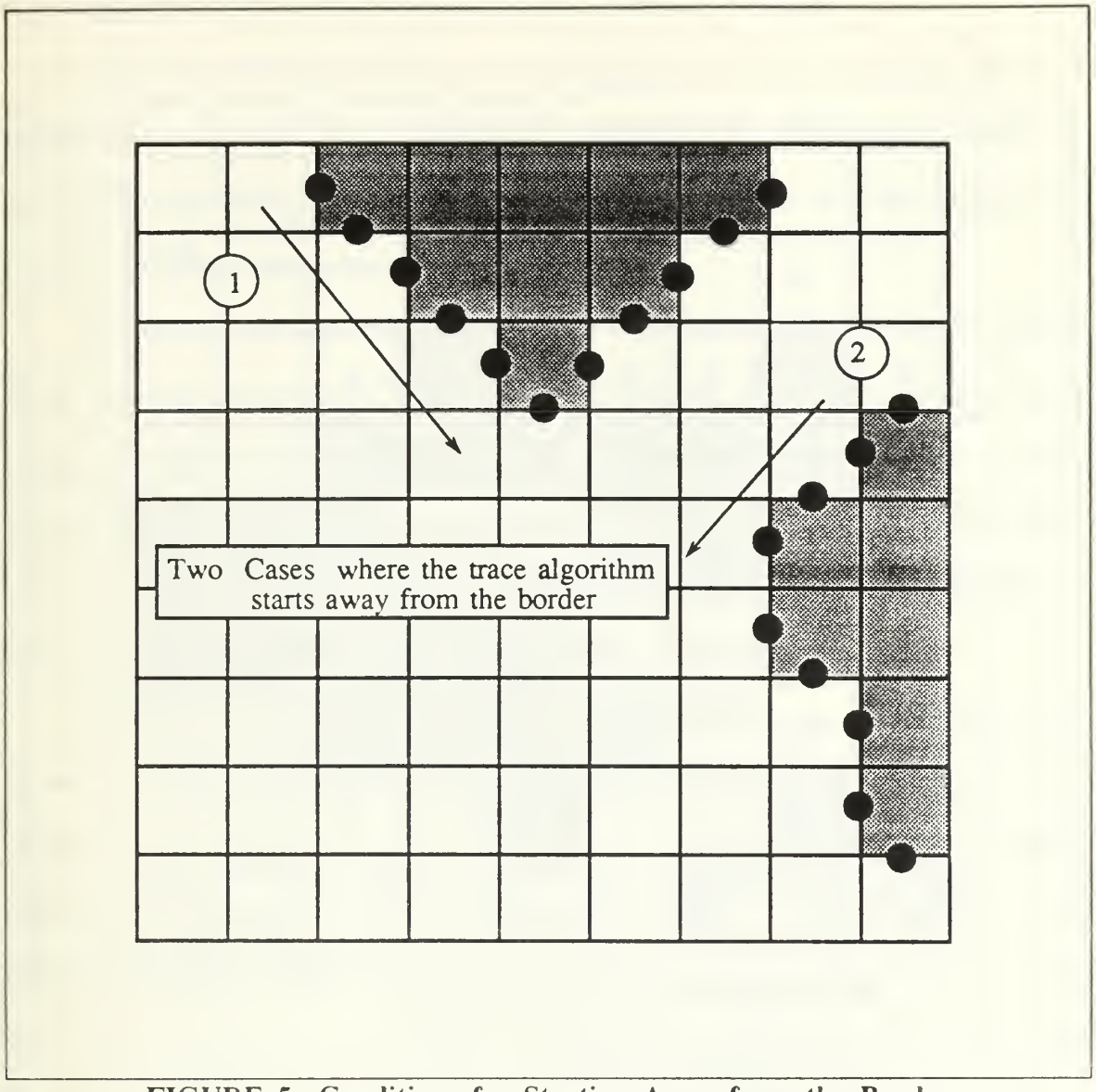


FIGURE 5 Conditions for Starting Away from the Border

The boundary-identification phase is limited to grids containing no more than 6400 input data points, and can handle up to 75 output regions of no more than 400 traced points. These constraints were selected to insure the program can handle grid squares with either many small regions, or grid squares with a few large regions. Although the normal cases for the boundary tracing algorithm have been tested, the

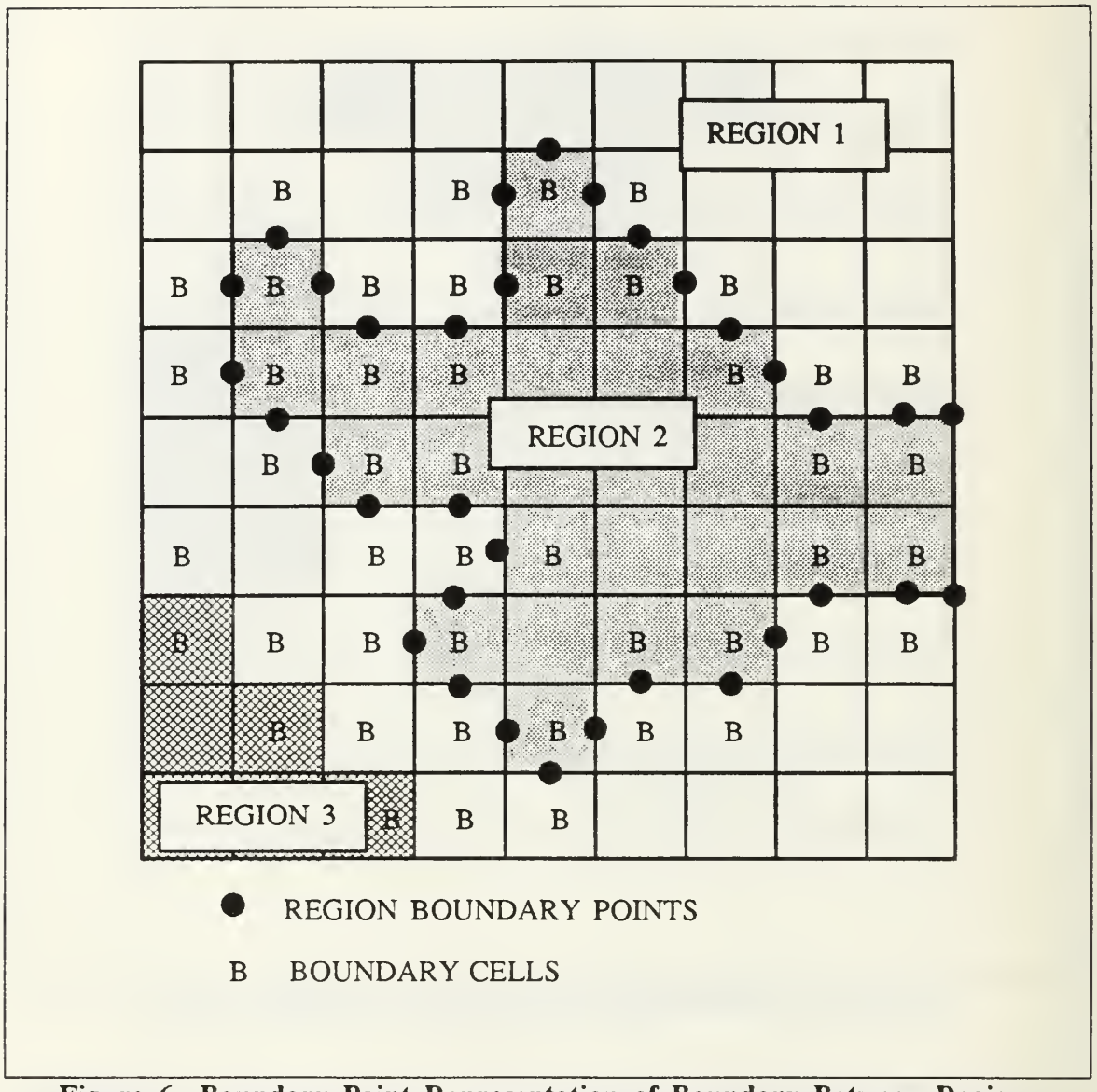


Figure 6 Boundary Point Representation of Boundary Between Regions

handling of all possible cases involving small regions, such as those containing 2 boundary cells near other regions, has not been verified to work, as in some unusual situations the tracing could skip points.

B. DESCRIPTION OF THE SPLIT-AND-MERGE PHASE

The split-and-merge phase of the program is written in the M-Prolog programming language for the Integrated Solutions (ISI) workstations. The list-processing capabilities of Prolog enables us to treat each region as a list of points and conduct operations on the members of the list.

The input to the split-and-merge phase is a Prolog fact for each region detected in the boundary-identification phase. Each fact contains an identification number of the region, the list of points for the region, the vegetation code of the region, and the total number of points in the region. See Figure 7. The output of the split-and-merge phase is a set of facts which contain vegetation codes and lists of the coordinates of each vertex for the polygonal homogeneous vegetation regions.

The major data structures used in the split-and-merge phase are lists, expressed as facts asserted throughout the phase. A global variable is created for the processing of each region. This contains the list of boundary points for that region; indexing based on the placement of the boundary points in the list is used to access it in the **linear least-squares** module. This saves allocated memory in terms of the statement table, global stack, and the main stack. A **segment** fact is asserted each time the linear least-squares fit is calculated for a line segment during the splitting process of the split-and-merge phase. See Figure 9. These facts are asserted and retracted often as the program seeks the proper combination of segments.

The split-and-merge phase contains five modules; the source code is contained in Appendix C. The **split-and-merge** module controls the phase using the built-in automatic backtracking of Prolog. The **linear least-squares** module calculates the fit

```
module regions.
/*$eject*/
body.
reg(0,[[0.0e0,22.5e0],
[1.0e0,21.0e0],
[1.0e0,21.5e0],
[2.0e0,21.5e0],
      •
      •
      •
[43.5e0,0.0e0]],3,62).
reg(1, [[0.0e0,72.5e0],
      •
      •
      •

/* The first fact describes region number 0.                    */
/* The region contains 62 data points.                          */
/* The value for the region is vegetation code 3.               */
```

Figure 7 Input Facts for Split-and-Merge Phase

of points in the vicinity of a line. The **list** module provides basic Prolog list-processing predicates. The **math** module provides math funtions such as the square root which are not provided in M-Prolog.[1] Finally, the **regions** module provides the data input from the boundary-identification phase.

The split-and-merge phase processes one pair of adjacent regions at a time, starting with pairs with non-zero vegetation codes. For each adjacent region, the list of boundary points from the two adjacent regions is intersected to produce a list of adjacent boundary points. To achieve clean boundaries between the two regions, no

[1] The source code for the **linear least-squares** module and most of the **math** and **list** modules were written by Professor Rowe. Prolog predicates written by Professor Rowe are marked with an asterisk in Appendix C.

```
asserta(segment(R,N1,N2,Fit12)).

/* R represents the region number of the segment.                    */
/* N1 and N2 are the vertices of the line segment.                   */
/* N1 and N2 are integer indices into the list of points             */
/*      for the region.                                              */
/* Fit12 is the linear least-squares fit of the ponts between N1 and */
/*      and N2 and the line defined by N1 and N2.                    */
```

Figure 8 Segment Facts

merging of line segments is allowed to use points from boundary point lists of differing region pairs. See Figure 8. The adjacent point list between the two regions will be handled twice for the two regions of the pair; however, the list will be split and merged identically in each case.

The splitting process follows the traditional divide-and-conquer algorithm [Ref 2:p. 282]. The linear least-squares fit is calculated for a segment, and a **segment** fact is asserted. If the fit is less than the splitting threshold, the segment is split into two new line segments which share a common point, the new "breakpoint" between them, and the old **segment** fact is retracted. This process continues recursively until there are no more **segment** facts to be split.

The calculation of the linear least-squares fit is done in the **linear least-squares** module using equation (8) from Chapter 2. The end points of each line segment define the line that will be tested for the fit. The use of the end points to define the line segment, rather than searching for the "best" line, enables us to connect adjacent line segments at exactly the breakpoints.

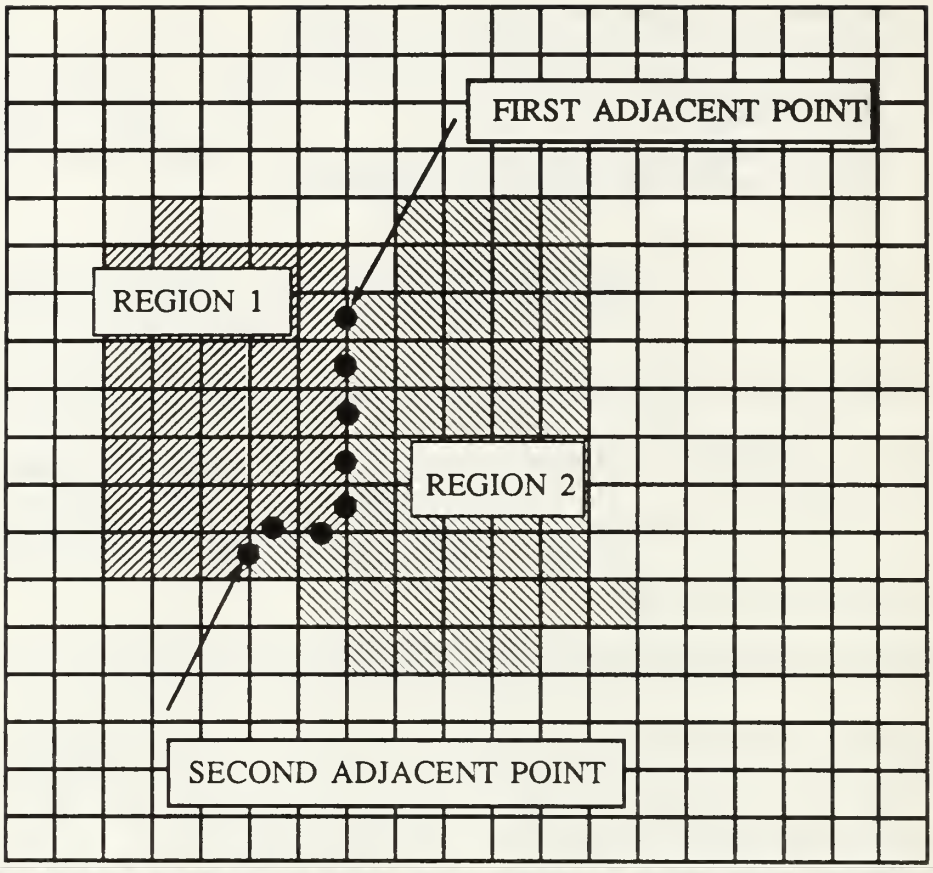


Figure 9 Adjacent Regions

The merge process examines each pair of line segments with a common index number (breakpoint) in the **segment** facts list. The segments are provisionally merged and the fit is calculated. If the fit is less than or equal to the merge threshold, the provisional merge is made permanent. A new **segment** fact is then asserted, and the

two original **segment** facts are then retracted. This process is continued until no more adjacent line segments can be merged.

The final process in the **linear least-squares** module of the split-and-merge phase takes the list of index numbers (representing the placement of the boundary points within the input list) and finds their actual x and y-coordinates. This shortened list is then bound to a **newregion** fact along with the value of the region.

V. RESULTS

A. PERFORMANCE MEASURES OF THE BOUNDARY-IDENTIFICATION PHASE

In test runs, the boundary-identification phase appears to properly define the boundaries of both convex and concave regions, including regions with holes. The maximum error for any point on the region boundary is equal to one half of the distance between the sample points, ignoring singular points. Since the input data points were evenly spaced every 12.5 meters, the maximum error at any point is equal to 6.25 meters.

The phase is memory-intensive. The program allocates 196800 bytes for the three arrays that are used to temporarily store information. This is more than three times the number of bytes required to process any of the test grid squares. Since we are not trying to optimize the code, we allocated more room than would probably be needed. Extensive use of file I/O, and the N squared nature of the boundary-identification algorithm, where N is the length of the square grid, slow the program to an average run time of 1 minute.

B. PERFORMANCE MEASURES OF THE SPLIT-AND-MERGE PHASE

The regions obtained from the split-and-merge phase appear to represent the input database without significant error. Examples of input terrain and the regions formed are contained in Appendix A. As expected, the fit threshold determines the

number of vertices for the output regions. Table 1 shows several thresholds and the resulting number of vertices for each region.

TABLE 1 EFFECT OF VARYING THE THRESHOLD ON THE NUMBER OF VERTICES PER REGION

REGION #	NUMBER OF POLYGON VERTICES			
	THRESHOLD 0.10	THRESHOLD 1.0	THRESHOLD 2.50	THRESHOLD 10.0
0	21	11	7	3
1	3	3	2	2
2	5	4	3	2
3	23	15	9	5
4	21	11	6	4
5	21	12	4	4
6	18	12	7	4

A threshold of 0.01 allows almost no merging. Conversely, a threshold of 10.0 reduces the regions to only a few vertices.

The split-and-merge phase is expensive in both time and space. Table 2 shows the maximum observed quantities for the Main Stack, Global Stack, Statement Table, Evaluations, cpu run time, and the real run time. The stacks and statement table are managed dynamically in our program, so the numbers vary through the running of the program. The higher numbers were obtained while processing large lists in the **linear**

TABLE 2 **MAXIMUM OBSERVED SIZE FOR M-PROLOG SYSTEM PARAMETERS IN THE SPLIT-AND-MERGE PHASE**

PARAMETERS	QUANTITY
MAIN STACK	1057 ITEMS
GLOBAL STACK	4544 ITEMS
STATEMENT TABLE	66,939 STATEMENTS
CPU RUN TIME	20 MINUTES 23 SECONDS
REAL RUN TIME	30 MINUTES

least-squares module. Prior to entering the **linear least-squares** module the Main Stack, Global Stack, and the Statement Table were 860 items, 660 items, and 60,836 statements respectively. These numbers only reflect the points in the program that we observed the system parameters. It is possible that these parameters could exceed the figures stated in Table 2 in other portions of the program.

VI. CONCLUSION

The primary objective of this work was twofold. First, we needed to show the feasibility of creating a polygonal region representation of terrain vegetation from digitized map data. Our program shows that this can be done. The program provides a suboptimal solution that visually appears to properly represent the terrain. But the split-and-merge phase of the program is time-consuming as the program searches for proper line segments. The many calculations involved implementing the fit calculations in M-Prolog is the main reason; other less mathematical methods for determining the fit of lines do exist and could easily replace the linear least-squares calculations without changing the rest of the split-and-merge phase.

The second objective of this work was to provide a working tool for path-planning research at the Naval Postgraduate School. This program will enable researchers to create polygonal regions of terrain vegetation.

The program only considers a 1km. by 1km. grid square. A more useful program for optimal path planning could consider much larger areas of terrain. This would probably require more efficient data storage in the boundary-identification phase and possibly optimizing and compiling the split-and-merge phase. An alternative and less costly technique would take the results from several adjacent grids and attempt to splice regions together that hit their respective borders.

Another topic for future research would be overlaying the results of this work with the three-dimensional planar-patch terrain models that Seung Hee Yee and

Professor Rowe developed. This would create partitioned homogeneous regions of constant slope as well as vegetation.

APPENDIX A TEST RESULTS

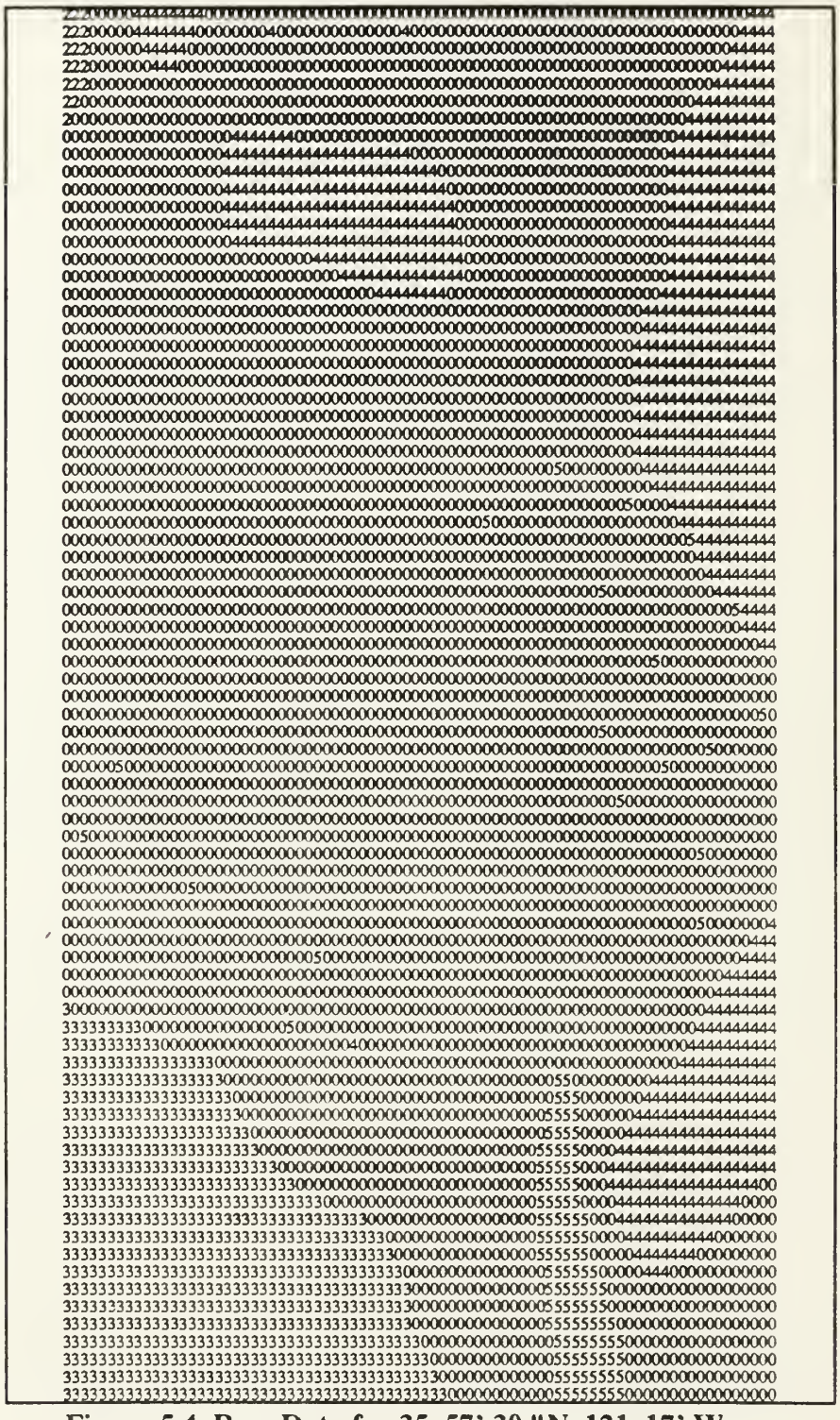
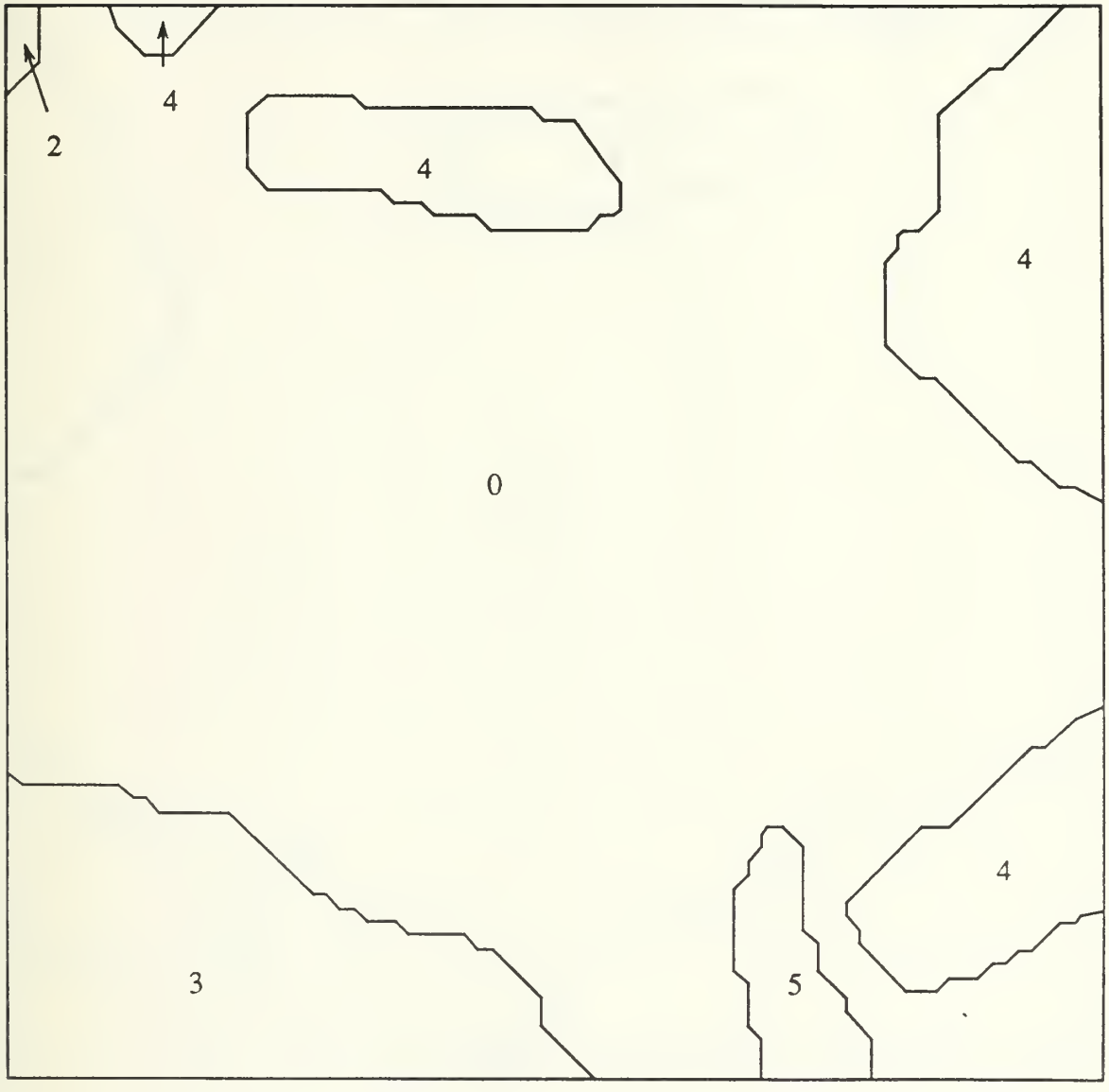


Figure 5.4 Raw Data for 35 57' 30 "N, 121 17' W

THRESHOLD EQUALS 0.01

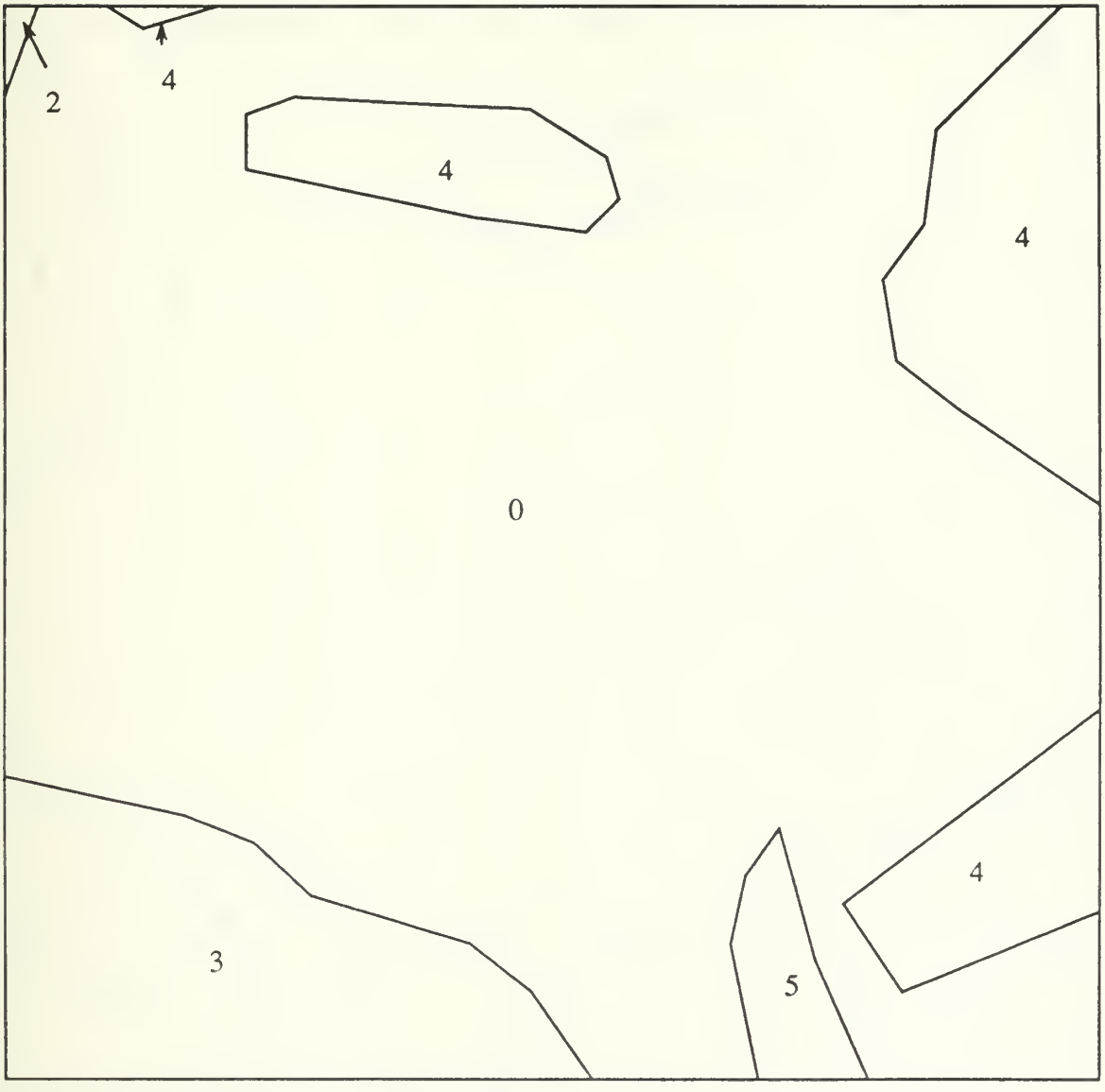


NUMBERS ARE VEGETATION CODES

THRESHOLD EQUALS 1.0

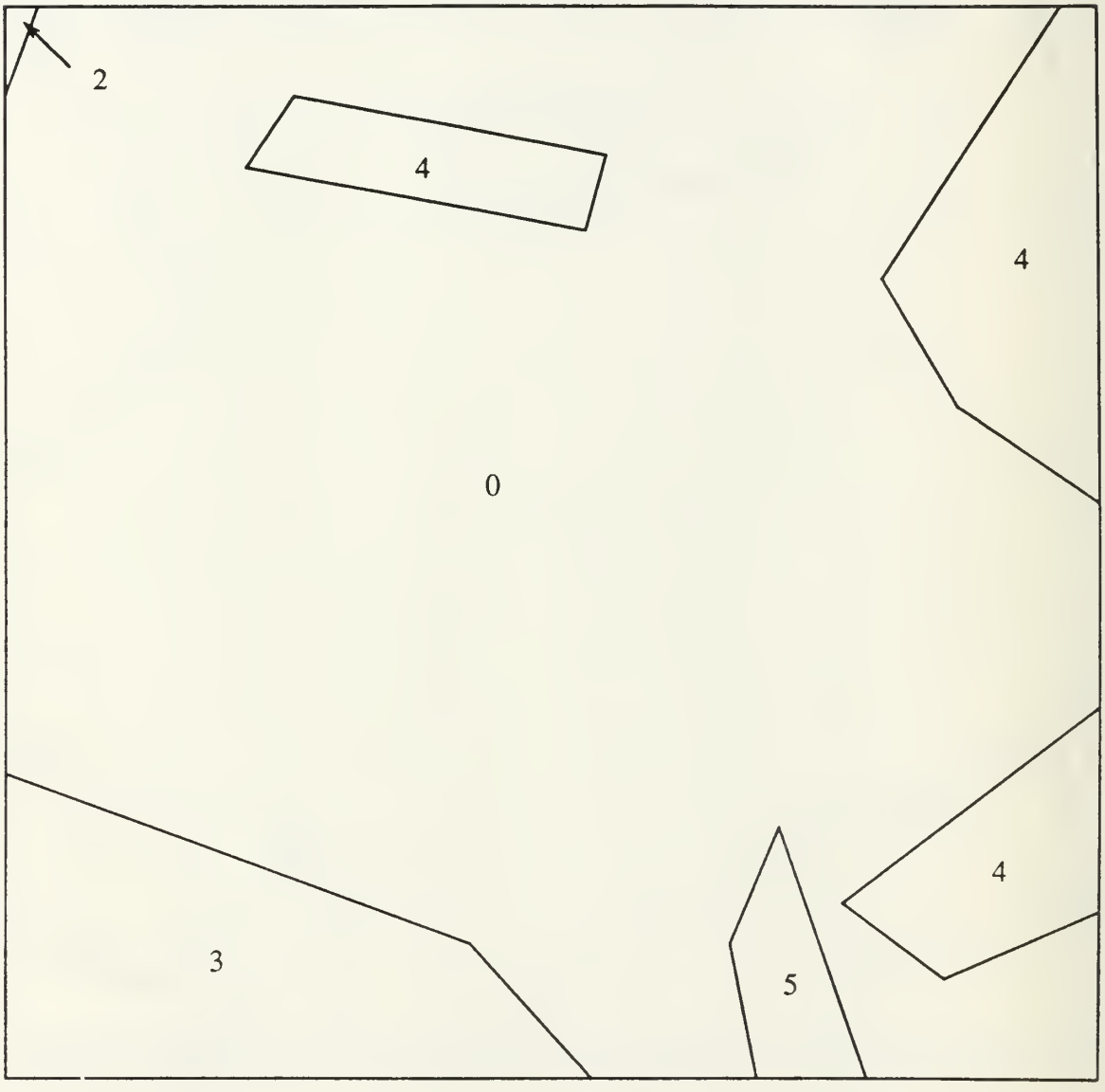
NUMBERS ARE VEGETATION CODES

THRESHOLD EQUALS 2.50



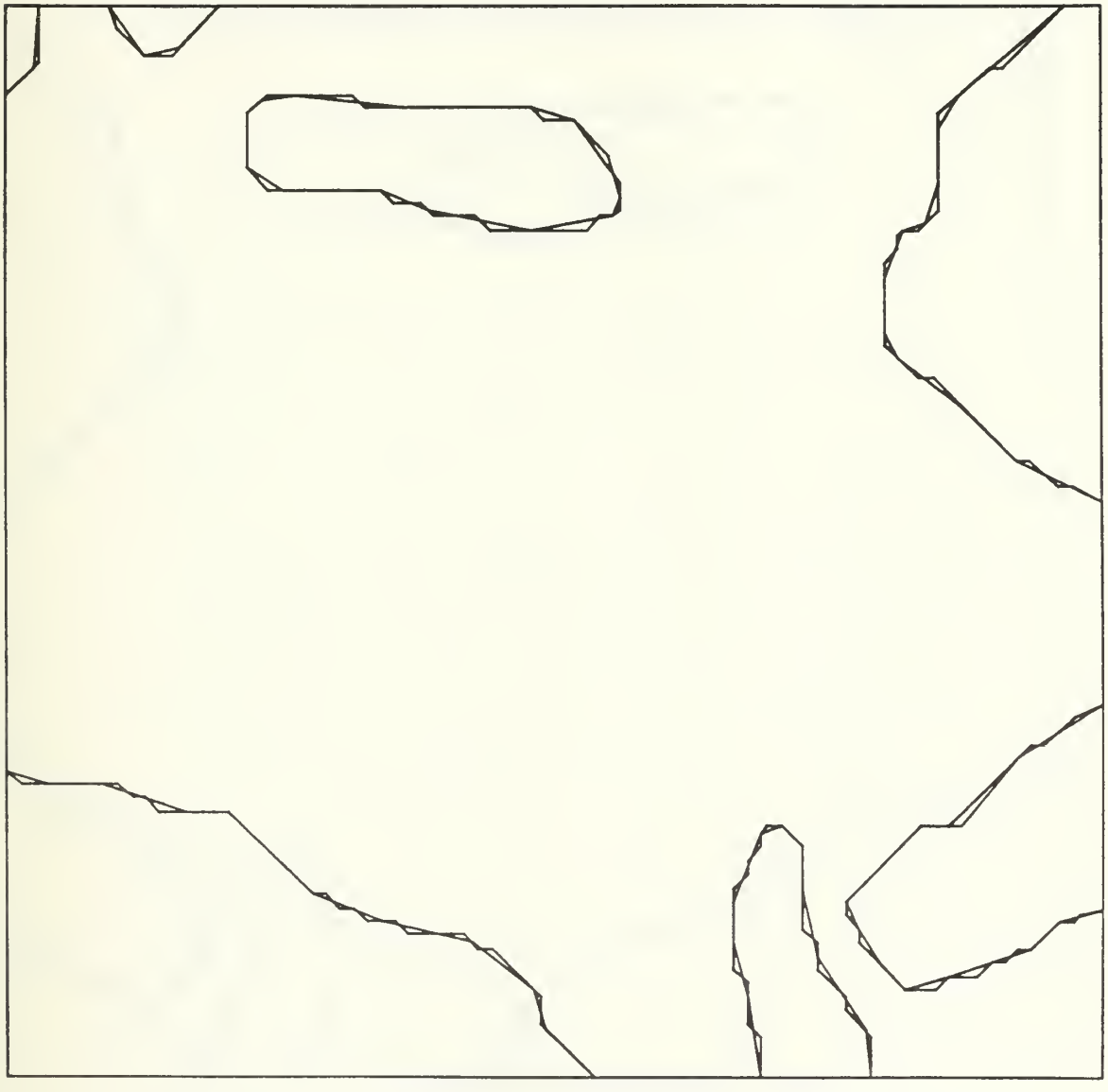
NUMBERS ARE VEGETATION CODES

THRESHOLD EQUALS 10.0

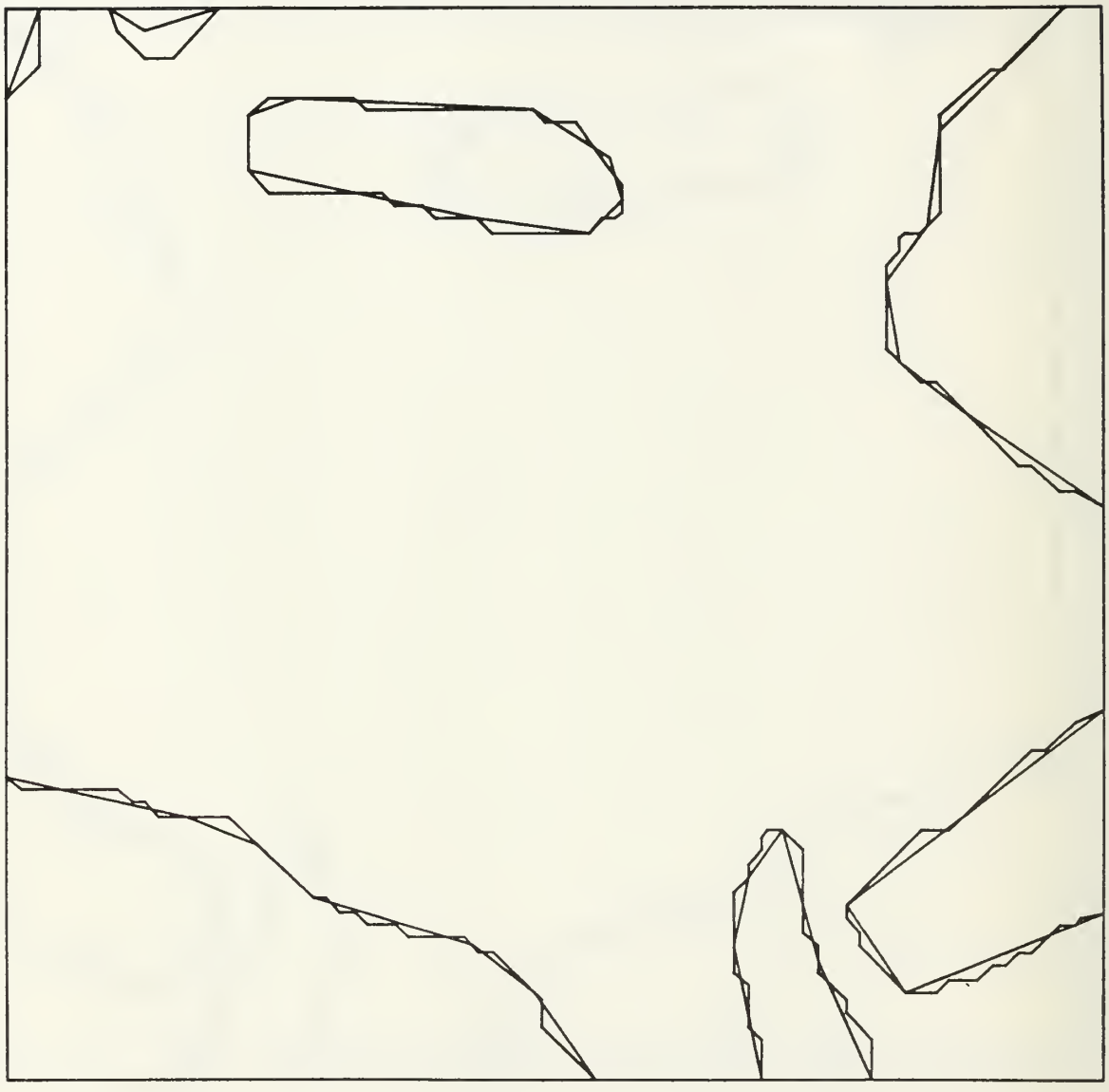


NUMBERS ARE VEGETATION CODES

THRESHOLDS 1.0 AND 0.01 SUPERIMPOSED



THRESHOLDS 0.01 AND 2.50 SUPERIMPOSED



THRESHOLDS 10.0 AND 0.01 SUPERIMPOSED

APPENDIX B SOURCE CODE FOR THE BOUNDARY-IDENTIFICATION PHASE

```
#include <ctype.h>
#include <stdio.h>
#include <math.h>
#include "header.h"

unsigned short vegarray[80][80];

unsigned short bndpts[6400][5];    /* 6400 pts x, y, value, boundary?, reg?  */

float regs[75][400][3]; /* 75 regions x 400 pts with x1, y1 in array coords */
                        /* and the region value.                            */

unsigned short pnt;
unsigned short veg;
unsigned short new_val;
unsigned short curr_dir;
unsigned short curr_pt;
unsigned short curr_x;
unsigned short curr_y;
unsigned short neighbor_x;
unsigned short neighbor_y;
unsigned short i,j,k;
boolean found_next;
boolean hit_border;
boolean stop_tracing;
boolean trace_counter;
boolean trace_border;
boolean start_counter;
int fdin;

main()
   {
   unsigned short x,y;
   int is_not_singlept();
   int single_adjacent_pt();
   int is_boundarypt();
   int can_extend();
   int adjacent_border();
```

```
    int curr_dir_diagonal();
    int start_backwards();
    fdin = open(infile,0);
/* creates the vegatation array 80 x 80 with values */
    for (x=0; x < 80; x++){
       for (y = 0; y < 80; y++){
          read(fdin, &veg, 2);
          vegarray[x][y] = veg - 48;
       }
    }
    pnt = 0;
    i = 0;  /* The current point in the second pass */
    j = 0;  /* The first four border regions are initiated w/out j */
    k = 0;  /* Next point # for current region being traced */
    for (x = 0; x < 80; x++){
       for (y = 0; y < 80; y++){
          if (x == 0 || x == 79 || y == 0 || y == 79) {
             bndpts[pnt][0] = x;
             bndpts[pnt][1] = y;
             bndpts[pnt][2] = vegarray[x][y];
             bndpts[pnt][3] = FALSE;
             bndpts[pnt][4] = TRUE;
          }
          else if (((vegarray[x][y] != vegarray[x][y+1] &&
                is_not_singlept(x,y+1)) ||
                (vegarray[x][y] != vegarray[x+1][y] &&
                is_not_singlept(x+1,y)) ||
                (vegarray[x][y] != vegarray[x][y-1] &&
                is_not_singlept(x,y-1)) ||
                (vegarray[x][y] != vegarray[x-1][y] &&
                is_not_singlept(x-1,y))) &&
                is_not_singlept(x,y)){
             bndpts[pnt][0] = x;
             bndpts[pnt][1] = y;
             bndpts[pnt][2] = vegarray[x][y];
             bndpts[pnt][3] = TRUE;
             bndpts[pnt][4] = FALSE;
          }  /*  if */
          else if (single_adjacent_pt(x,y)) {
             bndpts[pnt][0] = x;
             bndpts[pnt][1] = y;
             bndpts[pnt][2] = new_val;
             bndpts[pnt][3] = TRUE;
             bndpts[pnt][4] = FALSE;
          }  /* if */
        else {
```

```
            bndpts[pnt][0] = x;
            bndpts[pnt][1] = y;
            bndpts[pnt][2] = vegarray[x][y];
            bndpts[pnt][3] = FALSE;
            bndpts[pnt][4] = FALSE;
         }
         pnt++;
      } /* for y */
   } /* for x */
/* The second Pass through the data base starts here. This pass traces    */
/* the boundaries of each region by scanning through the entire DB looking */
/* for a boundary point. Once a boundary point is found it starts tracing */
/* around the region. The output is to the regs set of arrays.            */

   while (i < 6400) {
      curr_dir = North;
      k = 0;
      hit_border = FALSE;
      trace_border = FALSE;
      if (is_boundarypt()) {
         curr_pt = i;
         curr_x = bndpts[curr_pt][0];
         curr_y = bndpts[curr_pt][1];
         stop_tracing = FALSE;
         trace_counter = FALSE;
         start_counter = FALSE;
         if (adjacent_border())
            connect_border();
         if (start_backwards()) {
            start_counter = TRUE;
            trace_counter = TRUE;
            hit_border = TRUE;
            curr_dir = SEast;
         }
         do {
            if (start_counter) {
               right_front_neighbor();
               add_point();
               found_next = FALSE;
               while (!found_next) {
                  if (can_extend()) {
                     move_curr_dir();
                     found_next = TRUE;
                  }
                  else {
                     look_left_one();
```

```
            if (can_extend()) {
               move_curr_dir();
               found_next = TRUE;
            }
            else {
               front_neighbor();
               add_point();
               look_left_one();
            } /* inner else */
         }    /* outer else */
      } /* while not found next */
      start_counter = FALSE;
   } /* if start counter */
   else if (trace_counter && curr_dir_diagonal()) {
      right_rear_neighbor();
      add_point();
      look_right_two();
      found_next = FALSE;
      while (!found_next){
         if (can_extend()) {
            move_curr_dir();
            found_next = TRUE;
         }
         else {
            look_left_one();
            if (can_extend()) {
               move_curr_dir();
               found_next = TRUE;
            }
            else {
               front_neighbor();
               add_point();
               look_left_one();
            } /* inner else */
         } /* outer else */
      }    /* while not found_next  */
      if (curr_pt == i)
         continue;
   } /* if curr dir is diagonal and tracing counter clockwise */
   else if (trace_counter && ! curr_dir_diagonal() &&
       !trace_border) {
      right_neighbor();
      add_point();
      look_right_one();
      found_next = FALSE;
      while (!found_next) {
```

```
            if (can_extend()) {
               move_curr_dir();
               found_next = TRUE;
            }
            else {
               look_left_one();
               if (can_extend()) {
                  move_curr_dir();
                  found_next = TRUE;
               }
               else {
                  front_neighbor();
                  add_point();
                  look_left_one();
               } /* inner else */
            } /* outer else */
         } /* while not found_next */
         if (curr_pt == i)
            continue;
      } /* else curr_dir is not diagonal */
      else if (curr_dir_diagonal()) {
         left_rear_neighbor();
         add_point();
         look_left_two();
         found_next = FALSE;
         while (!found_next){
            if (can_extend()) {
               move_curr_dir();
               found_next = TRUE;
            }
            else {
               look_right_one();
               if (can_extend()) {
                  move_curr_dir();
                  found_next = TRUE;
               }
               else {
                  front_neighbor();
                  add_point();
                  look_right_one();
               }/* inner else */
            } /* outer else */
         }    /* while not found_next  */
         if (curr_pt == i)
            continue;
         } /* if curr dir is diagonal */
```

```
/* ********************** else curr_dir is not diagonal *********** */
                else if (!trace_border) {
                    left_neighbor();
                    add_point();
                    look_left_one();
                    found_next = FALSE;
                    while (!found_next) {
                        if (can_extend()) {
                            move_curr_dir();
                            found_next = TRUE;
                        }
                        else {
                            look_right_one();
                            if (can_extend()) {
                                move_curr_dir();
                                found_next = TRUE;
                            }
                            else {
                                front_neighbor();
                                add_point();
                                look_right_one();
                            } /* inner else */
                        } /* outer else */
                    } /* while not found_next */
                    if (curr_pt == i)
                        continue;
                } /* else curr_dir is not diagonal */
                if (adjacent_border()) {
                    bndpts[curr_pt][4] = TRUE;
                    connect_border();
                } /* if current point is adjacent border */
                } while ((!(curr_pt == i && !(trace_counter))) &&
                        !(stop_tracing));

/* If the region has the same start and end point make the last point the  */
/* same as the first point.                                                */
                if ((curr_pt == i) && (!hit_border))
                    duplicate_first_pt();

/* Set a flag at the end of each region so the output routine knows to stop */
                regs[j][k][0] = 9999;
                regs[j][k][1] = 9999;
                j++;  /* increment region number */
            } /* if is boundary point */
            i++;     /* increment point number that we are checking for bndy */
        } /* while i < 6400 */
```

```
    print_output_regions();

}  /* main for file bndpts.c */
/*******END of file BNDPTS.C ************************************/
```

File BNDUTIL.C

```
/* This file contains the necessary functions to execute the bndpts.c program.*/

#include <ctype.h>
#include <stdio.h>
#include "header.h"

extern unsigned short veg;
extern unsigned short new_val;
extern unsigned short curr_dir;
extern unsigned short curr_pt;
extern unsigned short curr_x;
extern unsigned short curr_y;
extern unsigned short i,j,k;
extern unsigned short neighbor_x;
extern unsigned short neighbor_y;
extern boolean trace_counter;
extern boolean trace_border;
extern boolean stop_tracing;
extern boolean hit_border;
extern unsigned short vegarray[80][80];
extern unsigned short bndpts[6400][5];
extern float regs[75][400][3];
extern int fdout;
extern int is_boundarypt();
extern int is_singlept();
extern int single_adjacent_pt();

/* ****************************************************************/
int is_not_singlept(x,y)
unsigned short x,y;
{
   return (vegarray[x][y] == vegarray[x+1][y] ||
      vegarray[x][y] == vegarray[x-1][y]  ||
      vegarray[x][y] == vegarray[x][y+1]  ||
      vegarray[x][y] == vegarray[x][y-1]);
}  /* function is_not_singlept  */

/*  ****************************************************************
*/
int single_adjacent_pt(x,y)
unsigned short x,y;
{
/* First find the maximum 4 way neighbor of the point.                */
   new_val = 0;
```

```
   if (vegarray[x + 1] [y] > vegarray[x][y + 1])
      new_val = vegarray[x+1][y];
   if (vegarray[x][y-1] > vegarray[x+1][y])
      new_val = vegarray[x][y-1];
   if (vegarray[x-1][y] > vegarray[x][y-1])
      new_val = vegarray[x-1][y];
   if (vegarray[x][y+1] > vegarray[x-1][y])
      new_val = vegarray[x][y+1];

/* Second, return true if the point has 4 way neighbors with at least two */
/* different values. Returns largest neighbors value as new_val.          */

   return(! is_not_singlept(x,y) &&
      (vegarray[x+1][y] != vegarray[x][y+1] ||
      vegarray[x+1][y] != vegarray[x][y-1] ||
      vegarray[x+1][y] != vegarray[x-1][y]));

}  /* function single_adjacent_pt */

/*  ************************************************************************
*/
/* A point is a boundary point if it is a boundary and it is not already */
/* part of a region and it does not have a value of zero.                */

int is_boundarypt()
{
   return (bndpts[i][3] == TRUE && bndpts[i][4] == FALSE &&
      bndpts[i][2] != 0);
}  /* function is_boundarypt */

/*  ************************************************************************
*/
/* A region is extended if the point in the curr direction is a boundary */
/* and the point in the curr direction is the same value as the current  */
/* point.                                                                */

int can_extend()
{
   switch (curr_dir)
   {
      case North:    return (bndpts[curr_pt + 1][3] == TRUE &&
                        bndpts[curr_pt + 1][2] == bndpts[curr_pt][2]);
      case NEast:    return (bndpts[curr_pt + 81][3] == TRUE &&
                        bndpts[curr_pt + 81][2] == bndpts[curr_pt][2]);
      case East:     return  (bndpts[curr_pt + 80][3] == TRUE &&
                        bndpts[curr_pt + 80][2] == bndpts[curr_pt][2]);
```

```
        case SEast:    return (bndpts[curr_pt + 79][3] == TRUE &&
                               bndpts[curr_pt + 79][2] == bndpts[curr_pt][2]);
        case South:    return (bndpts[curr_pt - 1][3] == TRUE &&
                               bndpts[curr_pt - 1][2] == bndpts[curr_pt][2]);
        case SWest:    return (bndpts[curr_pt - 81][3] == TRUE &&
                               bndpts[curr_pt - 81][2] == bndpts[curr_pt][2]);
        case West:     return (bndpts[curr_pt - 80][3] == TRUE &&
                               bndpts[curr_pt - 80][2] == bndpts[curr_pt][2]);
        case NWest:    return (bndpts[curr_pt - 79][3] == TRUE &&
                               bndpts[curr_pt - 79][2] == bndpts[curr_pt][2]);
        default:       printf("Current dir is not 0-7");
                       break;
        }  /* switch */
}  /* function     */

/* ************************************************************** */
/* Returns TRUE if the curr_pt is  adjacent to a border.          */

int adjacent_border()
{
   return (bndpts[curr_pt][0] == 1 || bndpts[curr_pt][0] == 78 ||
      bndpts[curr_pt][1] == 1 || bndpts[curr_pt][1] == 78 );

}  /* function adjacent_border */

/* ************************************************************** */
/* Returns TRUE if either of two conditions exist that cause the region */
/* to start tracing in the counterclockwise direction instead of the    */
/* clockwise direction.                                                 */

int start_backwards()
{
   return((bndpts[curr_pt][0] == 1 &&
   bndpts[curr_pt][2] == bndpts[curr_pt + 1][2]) ||
   bndpts[curr_pt][1] == 78);

}  /* function start_backwards */

/* ************************************************************** */
/* Changes the current direction one to the left.                 */

look_left_one()
{
   curr_dir = (curr_dir - 1) % 8;

}  /* function look_left_one  */
```

```
/* ************************************************************************/
/* Changes the current direction two to the left.                         */

look_left_two()
{
   curr_dir = (curr_dir - 2) % 8;

}  /* function look_left_two  */

/************************************************************************/
/* Changes the current direction one to the right.                        */

look_right_one()
{
   curr_dir = (curr_dir + 1) % 8;

} /* function look_right_one  */

/* ************************************************************************/
/* Changes the current direction two to the right.                        */

look_right_two()

{
   curr_dir = (curr_dir + 2) % 8;

}  /* function look_right_two  */

/* ************************************************************************/
/* Returns True if the current direction is diagonal (NEast, NWest,    */
/* SWest,SEast).                                                          */

int curr_dir_diagonal()
{
   return (curr_dir == NEast || curr_dir == SEast || curr_dir == SWest ||
      curr_dir == NWest);

} /* function curr_dir_diagonal */

/* ************************************************************************/
/* Connects the trace to the border. Places  the point where the trace    */
/* meets either of the four borders into into neighbor_x and neighbor_y. */

connect_border()
{
```

```
if (bndpts[curr_pt][0] == 1) {
   if (bndpts[curr_pt][2] != bndpts[curr_pt + 1][2]) {
      if (bndpts[curr_pt][2] == bndpts[curr_pt -79][2]) {
         regs[j][k][0] = 0.0;
         regs[j][k][1] = (float)(curr_y + 1.5);
         regs[j][k][2] = (float)(bndpts[curr_pt][2]);
         k++;
      }
      else if (bndpts[curr_pt][2] == bndpts[curr_pt -80][2]) {
         regs[j][k][0] = 0.0;
         regs[j][k][1] = (float)(curr_y + 0.5);
         regs[j][k][2] = (float)(bndpts[curr_pt][2]);
         k++;
      }
      else {
         regs[j][k][0] = 0.0;
         regs[j][k][1] = (float)(curr_y - 0.5);
         regs[j][k][2] = (float)(bndpts[curr_pt][2]);
         k++;
      }

      if (hit_border) {
       stop_tracing = TRUE;
      }
      else {
         hit_border = TRUE;
         curr_pt = i;
       curr_x = bndpts[curr_pt][0];
       curr_y = bndpts[curr_pt][1];
       curr_dir = East;
      }
   }
   else if (bndpts[curr_pt][2] != bndpts[curr_pt - 1][2]) {
      if (bndpts[curr_pt][2] == bndpts[curr_pt - 81][2]) {
         regs[j][k][0] = 0.0;
         regs[j][k][1] = (float)(curr_y - 1.5);
         regs[j][k][2] = (float)(bndpts[curr_pt][2]);
         k++;
      }
      else if (bndpts[curr_pt][2] == bndpts[curr_pt - 80][2]) {
         regs[j][k][0] = 0.0;
         regs[j][k][1] = (float)(curr_y - 0.5);
         regs[j][k][2] = (float)(bndpts[curr_pt][2]);
         k++;
      }
      else {
```

```
            regs[j][k][0] = 0.0;
            regs[j][k][1] = (float)(curr_y + 0.5);
            regs[j][k][2] = (float)(bndpts[curr_pt][2]);
            k++;
         } /* 3rd level else */
         if (hit_border)
            stop_tracing = TRUE;
         else {
            hit_border = TRUE;
            trace_counter = TRUE;
            curr_dir = South;
            curr_pt = i;
          curr_x = bndpts[curr_pt][0];
          curr_y = bndpts[curr_pt][1];
         }
      } /* 2nd level else */

      else if (trace_counter) {
            trace_border = TRUE;
          right_neighbor();
          add_point();
          curr_dir = North;
          move_curr_dir();
      }
      else if (!trace_counter) {
         trace_border = TRUE;
         left_neighbor();
         add_point();
         curr_dir = South;
         move_curr_dir();
      }
   } /* 1st level if */

   if (bndpts[curr_pt][0] == 78) {
      if (bndpts[curr_pt][2] != bndpts[curr_pt + 1][2]) {
         if (bndpts[curr_pt][2] == bndpts[curr_pt + 81][2]) {
            regs[j][k][0] = 79.0;
            regs[j][k][1] = (float)(curr_y + 1.5);
            regs[j][k][2] = (float)(bndpts[curr_pt][2]);
            k++;
         }
         else if (bndpts[curr_pt][2] == bndpts[curr_pt + 80][2]) {
            regs[j][k][0] = 79.0;
            regs[j][k][1] = (float)(curr_y + 0.5);
            regs[j][k][2] = (float)(bndpts[curr_pt][2]);
            k++;
```

```
    }
    else {
        regs[j][k][0] = 79.0;
        regs[j][k][1] = (float)(curr_y - 0.5);
        regs[j][k][2] = (float)(bndpts[curr_pt][2]);
        k++;
    }
    if (hit_border)
        stop_tracing = TRUE;
    else {
        hit_border = TRUE;
        trace_counter = TRUE;
        curr_dir = South;
        curr_pt = i;
     curr_x = bndpts[curr_pt][0];
     curr_y = bndpts[curr_pt][1];
    }
}
else if (bndpts[curr_pt][2] != bndpts[curr_pt - 1][2]) {
    if (bndpts[curr_pt][2] == bndpts[curr_pt + 79][2]) {
        regs[j][k][0] = 79.0;
        regs[j][k][1] = (float)(curr_y - 1.5);
        regs[j][k][2] = (float)(bndpts[curr_pt][2]);
        k++;
    }
    else if (bndpts[curr_pt][2] == bndpts[curr_pt + 80][2]) {
        regs[j][k][0] = 79.0;
        regs[j][k][1] = (float)(curr_y - 0.5);
        regs[j][k][2] = (float)(bndpts[curr_pt][2]);
        k++;
    }
    else {
        regs[j][k][0] = 79.0;
        regs[j][k][1] = (float)(curr_y + 0.5);
        regs[j][k][2] = (float)(bndpts[curr_pt][2]);
        k++;
    } /* 3rd level else */
    if (hit_border)
        stop_tracing = TRUE;
    else {
        hit_border = TRUE;
        trace_counter = TRUE;
        curr_dir = South;
        curr_pt = i;
     curr_x = bndpts[curr_pt][0];
     curr_y = bndpts[curr_pt][1];
```

```
        }
    }   /* 2nd level else */
    else if (trace_counter) {
        trace_border = TRUE;
        right_neighbor();
        add_point();
        curr_dir = South;
        move_curr_dir();
    }
    else if (!trace_counter) {
        trace_border = TRUE;
        left_neighbor();
        add_point();
        curr_dir = North;
        move_curr_dir();
    }
}   /* 1st level if */
if (bndpts[curr_pt][1] == 1) {
    if (bndpts[curr_pt][2] != bndpts[curr_pt -80][2]) {
        if (bndpts[curr_pt][2] == bndpts[curr_pt - 81][2]) {
            regs[j][k][0] = (float)(curr_x - 1.5);
            regs[j][k][1] = 0.0;
            regs[j][k][2] = (float)(bndpts[curr_pt][2]);
            k++;
        }
        else if (bndpts[curr_pt][2] == bndpts[curr_pt - 1][2]) {
            regs[j][k][0] = (float)(curr_x - 0.5);
            regs[j][k][1] = 0.0;
            regs[j][k][2] = (float)(bndpts[curr_pt][2]);
            k++;
        }
        else {
            regs[j][k][0] = (float)(curr_x + 0.5);
            regs[j][k][1] = 0.0;
            regs[j][k][2] = (float)(bndpts[curr_pt][2]);
            k++;
        }
        if (hit_border)
            stop_tracing = TRUE;
        else {
            hit_border = TRUE;
            trace_counter = TRUE;
            curr_dir = South;
            curr_pt = i;
         curr_x = bndpts[curr_pt][0];
         curr_y = bndpts[curr_pt][1];
```

```
    }
}

else if (bndpts[curr_pt][2] != bndpts[curr_pt + 80][2]) {
    if (bndpts[curr_pt][2] == bndpts[curr_pt + 79][2]) {
        regs[j][k][0] = (float)(curr_x + 1.5);
        regs[j][k][1] = 0.0;
        regs[j][k][2] = (float)(bndpts[curr_pt][2]);
        k++;
    }
    else if (bndpts[curr_pt][2] == bndpts[curr_pt - 1][2]) {
        regs[j][k][0] = (float)(curr_x + 0.5);
        regs[j][k][1] = 0.0;
        regs[j][k][2] = (float)(bndpts[curr_pt][2]);
        k++;
    }
    else {
        regs[j][k][0] = (float)(curr_x - 0.5);
     regs[j][k][1] = 0.0;
        regs[j][k][2] = (float)(bndpts[curr_pt][2]);
     k++;
    }  /* 3rd level else */
    if (hit_border){
        stop_tracing = TRUE;
    }
    else {
        hit_border = TRUE;
        trace_counter = TRUE;
        curr_dir = South;
        curr_pt = i;
     curr_x = bndpts[curr_pt][0];
     curr_y = bndpts[curr_pt][1];
    }
}    /* 2nd level else */
else if (trace_counter) {
    trace_border = TRUE;
    right_neighbor();
    add_point();
    curr_dir = West;
    move_curr_dir();
}
else if (!trace_counter) {
    trace_border = TRUE;
    left_neighbor();
    add_point();
    curr_dir = East;
```

```
            move_curr_dir();
        }
    }       /* 1st level else */
    if (bndpts[curr_pt][1] == 78) {
        if (bndpts[curr_pt][2] != bndpts[curr_pt - 80][2]) {
            if (bndpts[curr_pt][2] == bndpts[curr_pt - 79][2]) {
                regs[j][k][0] = (float)(curr_x - 1.5);
                regs[j][k][1] = 79.0;
                regs[j][k][2] = (float)(bndpts[curr_pt][2]);
                k++;
            }
            else if (bndpts[curr_pt][2] == bndpts[curr_pt + 1][2]) {
                regs[j][k][0] = (float)(curr_x - 0.5);
                regs[j][k][1] = 79.0;
                regs[j][k][2] = (float)(bndpts[curr_pt][2]);
                k++;
            }
            else {
                regs[j][k][0] = (float)(curr_x + 0.5);
                regs[j][k][1] = 79.0;
                regs[j][k][2] = (float)(bndpts[curr_pt][2]);
                k++;
            }

            if (hit_border)
                stop_tracing = TRUE;
            else {
                hit_border = TRUE;
                trace_counter = TRUE;
                curr_dir = South;
                curr_pt = i;
             curr_x = bndpts[curr_pt][0];
             curr_y = bndpts[curr_pt][1];
            }
        }

        else if (bndpts[curr_pt][2] != bndpts[curr_pt + 80][2]) {
            if (bndpts[curr_pt][2] == bndpts[curr_pt + 81][2]) {
                regs[j][k][0] = (float)(curr_x + 1.5);
                regs[j][k][1] = 79.0;
                regs[j][k][2] = (float)(bndpts[curr_pt][2]);
                k++;
            }
            else if (bndpts[curr_pt][2] == bndpts[curr_pt + 1][2]) {
                regs[j][k][0] = (float)(curr_x + 0.5);
                regs[j][k][1] = 79.0;
```

```
            regs[j][k][2] = (float)(bndpts[curr_pt][2]);
            k++;
         }
         else {
            regs[j][k][0] = (float)(curr_x - 0.5);
            regs[j][k][1] = 79.0;
            regs[j][k][2] = (float)(bndpts[curr_pt][2]);
            k++;
         } /* 3rd level else */

         if (hit_border)
            stop_tracing = TRUE;
         else {
            hit_border = TRUE;
            trace_counter = TRUE;
            curr_dir = South;
            curr_pt = i;
          curr_x = bndpts[curr_pt][0];
          curr_y = bndpts[curr_pt][1];
         }
      }   /* 2nd level else */
      else if (trace_counter) {
         trace_border = TRUE;
         right_neighbor();
         add_point();
         curr_dir = East;
         move_curr_dir();
      }
      else if (!trace_counter) {
         trace_border = TRUE;
         left_neighbor();
         add_point();
         curr_dir = West;
         move_curr_dir();
      }

   } /* 1st level if */
   hit_border = TRUE;

} /* function connect_border */

/*************************************************************************/

/* Inserts a point to the current region output database, and increments */
/* the point count.                                                      */
```

```
add_point()
{
   regs[j][k][0] = (float) (curr_x + neighbor_x)/2;
   regs[j][k][1] = (float) (curr_y + neighbor_y)/2;
   regs[j][k][2] = (float) (bndpts[curr_pt][2]);
   k++; /* increment next point to be added to the region */

} /* function add_point */

/* ************************************************************************/
/* Inserts a point to the current region output database, and increments */
/* the point count. This is used when the region is closed, and the     */
/* first point is the same as the last point.                           */

duplicate_first_pt()
{
   regs[j][k][0] = regs[j][0][0];
   regs[j][k][1] = regs[j][0][1];
   k++; /* increment next point to be added to the region */

}  /* function duplicate_first_pt */

/************************************************************************/

/* Gets the x,y of the point to the left of the curr point.             */
/* If we are tracing counterclockwise,:we get the x,y of point to the right  */

left_neighbor()

{

   switch (curr_dir)
   {
      case North :   neighbor_x = curr_x - 1;
                     neighbor_y = curr_y;
                     break;
      case NEast :   neighbor_x = curr_x -1;
                     neighbor_y = curr_y +1;
                     break;
      case East :    neighbor_x = curr_x;
                     neighbor_y = curr_y + 1;
                     break;
      case  SEast :  neighbor_x = curr_x + 1;
                     neighbor_y = curr_y + 1;
                     break;
      case South :   neighbor_x = curr_x +1;
```

```
                    neighbor_y = curr_y;
                    break;
        case SWest :  neighbor_x = curr_x + 1;
                    neighbor_y = curr_y - 1;
                    break;
        case West :   neighbor_x = curr_x;
                    neighbor_y = curr_y - 1;
                    break;
        case NWest :  neighbor_x = curr_x - 1;
                    neighbor_y = curr_y - 1;
                    break;
        default:    printf("Error curr-dir != 0-7");
                    break;

        } /* switch */

} /* function   */

/*****************************************************************/
/* Gets  the x,y of the left rear neigbor of the current point.    */

left_rear_neighbor()
{
   switch (curr_dir)
   {
        case North :  neighbor_x = curr_x - 1;
                    neighbor_y = curr_y - 1;
                    break;
        case NEast :  neighbor_x = curr_x -1;
                    neighbor_y = curr_y;
                    break;
        case East :   neighbor_x = curr_x - 1;
                    neighbor_y = curr_y + 1;
                    break;
        case SEast :  neighbor_x = curr_x;
                    neighbor_y = curr_y + 1;
                    break;
        case South :  neighbor_x = curr_x + 1;
                    neighbor_y = curr_y + 1;
                    break;
        case SWest :  neighbor_x = curr_x + 1;
                    neighbor_y = curr_y;
                    break;
        case West :   neighbor_x = curr_x + 1;
                    neighbor_y = curr_y - 1;
                    break;
```

```
        case NWest :    neighbor_x = curr_x;
                        neighbor_y = curr_y - 1;
                        break;
        default:        printf("Error curr-dir != 0-7");
                        break;
        } /* switch */

} /* function       /*

/*****************************************************************/
/* Gets the x,y of the left front neighbor of the current point.  */

left_front_neighbor()
{
   switch (curr_dir)
   {
      case North :   neighbor_x = curr_x - 1;
                     neighbor_y = curr_y + 1;
                     break;
      case NEast :   neighbor_x = curr_x;
                     neighbor_y = curr_y +1;
                     break;
      case East :    neighbor_x = curr_x + 1;
                     neighbor_y = curr_y + 1;
                     break;
      case SEast :   neighbor_x = curr_x + 1;
                     neighbor_y = curr_y;
                     break;
      case South :   neighbor_x = curr_x +1;
                     neighbor_y = curr_y - 1;
                     break;
      case SWest :    neighbor_x = curr_x;
                     neighbor_y = curr_y - 1;
                     break;
      case West :    neighbor_x = curr_x - 1;
                     neighbor_y = curr_y - 1;
                     break;
      case NWest :    neighbor_x = curr_x - 1;
                     neighbor_y = curr_y;
                     break;
      default:       printf("Error curr-dir != 0-7");
                     break;

   } /* switch  */

} /* function   */
```

```
/*****************************************************************/
/* Gets the x,y of the front neighbor of the current point.      */

front_neighbor()

{
   if (! trace_counter) {
      switch (curr_dir)
      {
         case North :   neighbor_x = curr_x;
                        neighbor_y = curr_y + 1;
                        break;
         case NEast :   neighbor_x = curr_x + 1;
                        neighbor_y = curr_y + 1;
                        break;
         case East :    neighbor_x = curr_x + 1;
                        neighbor_y = curr_y;
                        break;
         case SEast :   neighbor_x = curr_x + 1;
                        neighbor_y = curr_y - 1;
                        break;
         case South :   neighbor_x = curr_x;
                        neighbor_y = curr_y - 1;
                        break;
         case SWest :   neighbor_x = curr_x - 1;
                        neighbor_y = curr_y - 1;
                        break;
         case West :    neighbor_x = curr_x - 1;
                        neighbor_y = curr_y;
                        break;
         case NWest :   neighbor_x = curr_x - 1;
                        neighbor_y = curr_y + 1;
                        break;
         default:       printf("Error curr-dir != 0-7");
                        break;
      } /* switch  */
   } /* if trace left */
   else {
      switch (curr_dir)
      {
         case North :   neighbor_x = curr_x;
                        neighbor_y = curr_y + 1;
                        break;
         case NEast :   neighbor_x = curr_x + 1;
                        neighbor_y = curr_y + 1;
```

```
                        break;
        case East :     neighbor_x = curr_x + 1;
                        neighbor_y = curr_y;
                        break;
        case SEast :    neighbor_x = curr_x + 1;
                        neighbor_y = curr_y - 1;
                        break;
        case South :    neighbor_x = curr_x;
                        neighbor_y = curr_y - 1;
                        break;
        case SWest :    neighbor_x = curr_x - 1;
                        neighbor_y = curr_y - 1;
                        break;
        case West :     neighbor_x = curr_x - 1;
                        neighbor_y = curr_y;
                        break;
        case NWest :    neighbor_x = curr_x - 1;
                        neighbor_y = curr_y + 1;
                        break;
        default:        printf("Error curr-dir != 0-7");
                        break;
     } /* switch  */
   } /* else trace right */
} /* function        */

/*************************************************************************/

/* Gets the x,y of the point to the left of the curr point.              */
/* If we are tracing counterclockwise, we get the x,y of point to the right  */

right_neighbor()
{
   switch (curr_dir)
   {
      case North :   neighbor_x = curr_x + 1;
                     neighbor_y = curr_y;
                     break;
      case NEast :   neighbor_x = curr_x +1;
                     neighbor_y = curr_y -1;
                     break;
      case East :    neighbor_x = curr_x;
                     neighbor_y = curr_y - 1;
                     break;
      case  SEast :  neighbor_x = curr_x - 1;
                     neighbor_y = curr_y - 1;
                     break;
```

```
      case South :   neighbor_x = curr_x -1;
                     neighbor_y = curr_y;
                     break;
      case SWest :    neighbor_x = curr_x - 1;
                     neighbor_y = curr_y + 1;
                     break;
      case West :     neighbor_x = curr_x;
                     neighbor_y = curr_y + 1;
                     break;
      case NWest :    neighbor_x = curr_x + 1;
                     neighbor_y = curr_y + 1;
                     break;
      default:       printf("Error curr-dir != 0-7");
                     break;

   } /* switch */
} /* function */

/*************************************************************************/

/* Gets the x,y of the point to the right front of the curr point.      */

right_front_neighbor()
{
   switch (curr_dir)
   {
      case North :   neighbor_x = curr_x + 1;
                     neighbor_y = curr_y + 1;
                     break;
      case NEast :    neighbor_x = curr_x + 1;
                     neighbor_y = curr_y;
                     break;
      case East :    neighbor_x = curr_x + 1;
                     neighbor_y = curr_y - 1;
                     break;
      case  SEast :  neighbor_x = curr_x;
                     neighbor_y = curr_y - 1;
                     break;
      case South :   neighbor_x = curr_x - 1;
                     neighbor_y = curr_y - 1;
                     break;
      case SWest :    neighbor_x = curr_x - 1;
                     neighbor_y = curr_y;
                     break;
      case West :     neighbor_x = curr_x - 1;
                     neighbor_y = curr_y + 1;
```

```
                    break;
        case NWest :   neighbor_x = curr_x;
                    neighbor_y = curr_y + 1;
                    break;
        default:    printf("Error curr-dir != 0-7");
                    break;

    } /* switch */
} /* function */

/*************************************************************************/

/* Gets the x,y of the point to the right rear of the curr point.      */

right_rear_neighbor()
{
    switch (curr_dir)
    {
        case North :   neighbor_x = curr_x + 1;
                    neighbor_y = curr_y - 1;
                    break;
        case NEast :   neighbor_x = curr_x;
                    neighbor_y = curr_y - 1;
                    break;
        case East :    neighbor_x = curr_x - 1;
                    neighbor_y = curr_y - 1;
                    break;
        case  SEast :  neighbor_x = curr_x - 1;
                    neighbor_y = curr_y;
                    break;
        case South :   neighbor_x = curr_x - 1;
                    neighbor_y = curr_y + 1;
                    break;
        case SWest :   neighbor_x = curr_x;
                    neighbor_y = curr_y + 1;
                    break;
        case West :    neighbor_x = curr_x + 1;
                    neighbor_y = curr_y + 1;
                    break;
        case NWest :   neighbor_x = curr_x + 1;
                    neighbor_y = curr_y;
                    break;
        default:    printf("Error curr-dir != 0-7");
                    break;
    } /* switch */
} /* function */
```

```
/*************************************************************************/

/*  Marks the current point as used in a region, and changes the current */
/*  point in the direction of the current direction. Changes the curr_x, */
/*  and curr_y to correspond with the new current point.                 */

move_curr_dir()
{
   bndpts[curr_pt][4] = TRUE;
   switch (curr_dir)
   {
      case North :   curr_pt = curr_pt + 1;
                     break;
      case NEast :   curr_pt = curr_pt + 81;
                     break;
      case East :    curr_pt = curr_pt + 80;
                     break;

      case SEast :   curr_pt = curr_pt + 79;
                     break;
      case South :   curr_pt = curr_pt - 1;
                     break;
      case SWest :   curr_pt = curr_pt - 81;
                     break;
      case West :    curr_pt = curr_pt - 80;
                     break;
      case NWest :   curr_pt = curr_pt - 79;
                     break;
      default :      printf("Curr_dir != 0-7");
                     break;
   }
   curr_x = bndpts[curr_pt][0];
   curr_y = bndpts[curr_pt][1];

}  /* function move_curr_dir */

/*************************************************************************/

print_output_regions()
{
   FILE *fdout;
   FILE *foutfig;
   unsigned short r;
   unsigned short s;
   int temp;
```

```
    fdout = fopen(outfile, "w+");
    foutfig = fopen(outfig, "w+");
    if (fdout == NULL) {
    printf("Have file opening problem");
    }
    if (foutfig == NULL) {
    printf("Have file opening problem");
    }

/* Writes to file and screen the x and y value for each of the regions.      */
/* The stopping conditions of the loops are when the number of regions       */
/* generated is reached, and for each region when the flag 9999 is reached. */
/* This function assumes it knows the number of regions (j -1) and the       */
/* array element after the last point in each region is flagged.             */

    fprintf(fdout,"module regions.\n");
    fprintf(fdout,"/*$eject*/\n");
    fprintf(fdout,"body.\n");
    for (r = 0; r < j; r++) {
       fprintf(fdout,"reg(%hu, [",r);
       s = 0;
       do {

/* Convert to inches only for the foutfig file.                              */

          printf("%4hu %4f %4f\n", r,regs[r][s][0],regs[r][s][1]);
          fprintf(fdout,"[%.1fe0, %.1fe0]",regs[r][s][0],
             regs[r][s][1]);
          regs[r][s][0] = (regs[r][s][0] / 10);
          regs[r][s][1] = (regs[r][s][1] / 10);
          fprintf(foutfig,"%4f %4f\n ",regs[r][s][0],
             regs[r][s][1]);
          fprintf(foutfig,"%4f %4f\n ",regs[r][s][0],
             regs[r][s][1]);
          s++;
          if (regs[r][s][0] != 9999.000000) {
             fprintf(fdout,",\n");
          }
       } while ((regs[r][s][0] != 9999.000000) ||
                (regs[r][s][1] != 9999.0000000));
       temp = (int)(regs[r][0][2]);
       fprintf(fdout,"],%hu,%hu).\n",temp,s);
    }   /* for r  */
    fprintf(fdout,"numregions(%hu).\n",r);
    fprintf(fdout,"endmod.\n");
    fclose(fdout);
```

```
    fclose(foutfig);
}  /* Print the output of the regions.     */
/* End of BNDUTIL.C ***************************************************/
```

File header.h

```
/* This file sets constants and determines the path for the input and */
/* output file for the bndpts.c program.                              */

#define infile "/work/wade/Thesis/C/rawdata2.h"
#define outfile "/work/wade/Thesis/C/regiondata2.pro"
#define outfig "/work/wade/Thesis/C/regiondata2.fig"
#define boolean int
#define FALSE 0
#define TRUE !(FALSE)
#define North 0
#define NEast 1
#define East 2
#define SEast 3
#define South 4
#define SWest 5
#define West 6
#define NWest 7

/********END of file header.h ***************************/
```

File vdb2.c

```
/* This file is the main program for getting a 1 Km by 1 Km grid square */
/* from the DTED data base. The program was run on the Silicon Graphics */
/* Computer. The program requires file "files.h" to run. The program is */
/* a variant of a widely used program in the department to get the data */

#include "stdio.h"
#include "ctype.h"
#include "math.h"
#include "files.h"

char infile[50]= MASTER_DMA_DTED_FILE;
int  fdin;

main()
{
 int    fdout,x,z,r,c,i,j,utmx,utmz;
 int    off,ewerror=1,nserror=1,doswap=1;
 char   s,swap,outfile[50],utmew[5],utmns[5],edbase[20402],temp;
 unsigned short raw,elev,veg,massaged;
 short MAXUTMEW=659,MAXUTMNS=849,DATAPTS=11;

 system("clear");
 printf("THIS PROGRAM WILL CREATE A TERRAIN DATABASE.\n");
 printf("THE SOURCE DATABASE IS A DEFENSE MAPPING AGENCY
DIGITAL\n");
 printf("TERRAIN ELEVATION FILE FOR A 36 KM BY 35 KM REGION OF\n");
 printf("FORT HUNTER LIGGET, CA. AND VICINITY.\n");
 printf("THE OUTPUT FILE IS A 1 KM BY 1 KM SUBSET OF THE ENTIRE\n");

 printf("REGION, AND WILL BE STORED IN THE FORMAT REQUIRED\n");
 printf("              100.0 meter resolution\n");
 printf("              80 x 80 data points\n");
 printf("              storage in z major order\n");
 printf("YOU MUST ENTER THE UTM COORDINATES OF THE SOUTHWEST\n");

 printf("CORNER OF THE SUBSET REGION YOU WANT EXTRACTED.\n");
 printf("        ENTER A CARRIAGE RETURN TO CONTINUE...");


 while(ewerror) {
  system("clear");
  printf("          VALID EW COORDINATES ARE IN THE RANGES:\n");
  printf("              EAST-WEST (UTM EW):  410 to %d\n\n",MAXUTMEW);
  printf("                   ********************\n");
```

```
  printf("                    *                   *\n");
  printf("                    *                   *\n");
  printf("                    *         N         *\n");
  printf("                    *         |         *\n");
  printf("                    *      W-----E      *\n");
  printf("                    *         |         *\n");
  printf("                    *         S         *\n");
  printf("                    *                   *\n");
  printf("                    *                   *\n");
  printf("                    ********************\n");
  printf("                    |                   |\n");
  printf("                  410                   %d\n",MAXUTMEW);
  printf("\n\nENTER the UTM EW coordinate of the southwest corner\n");
  printf("    ( Enter an integer value X:    410 <= X <= %d )\n",MAXUTMEW);
  printf("                    X ? ==>");
  scanf("%d",&utmx);
  if( (utmx>=410) && (utmx<=MAXUTMEW) ) ewerror=0;
 }
 while(nserror) {
  system("clear");
  printf("        VALID NS COORDINATES ARE IN THE RANGES:\n");
  printf("          NORTH-SOUTH (UTM NS):  600 to %d\n\n",MAXUTMNS);
  printf("              %3d- ********************\n",MAXUTMNS);
  printf("                    *                   *\n");
  printf("                    *                   *\n");
  printf("                    *         N         *\n");
  printf("                    *         |         *\n");
  printf("                    *      W-----E      *\n");
  printf("                    *         |         *\n");
  printf("                    *         S         *\n");
  printf("                    *                   *\n");
  printf("                    *                   *\n");
  printf("                600- ********************\n");
  printf("\n\nENTER the UTM NS coordinate of the southwest corner\n");
  printf("    ( Enter an integer value Y:    600 <= Z <= %d )\n",MAXUTMNS);
  printf("                    Z ? ==>");
  scanf("%d",&utmz);
  if( (utmz>=600)&&(utmz<=MAXUTMNS) ) nserror=0;
 }

 system("clear");
 printf("DO YOU WISH ELEVATION DATA WORDS TO HAVE THEIR BYTES
SWAPPED?\n");
 printf("          ENTER 'y' ( YES: SWAP BYTES   )\n");
 printf("          ENTER 'n' ( NO: NO BYTE SWAP )\n");
 printf("          SWAP BYTES ==> ?");
```

```
 swap = toupper(getchar());
 while( (swap != 'Y') && (swap != 'N') ) swap=toupper(getchar());
 if(swap=='N') doswap=0;
 system("clear");
 sprintf(utmew,"%3d",utmx);
 sprintf(utmns,"%3d",utmz);
 strcpy(outfile,OUTPUTFILE);
 if(doswap)      strcat(outfile,"swap");
 printf("\nCreating database for a 1km x 1km region, southwest corner");
 printf("\nat %d - %d.  Database will consist of elevation data",utmx,utmz);
 printf("\nfor %d x %d points at 12.5 meter resolution.",DATAPTS,DATAPTS);
 printf("\nDatabase filename is %s\n",outfile);
 fdin = open(infile,0);
 fdout= creat(outfile,0644);


 r = (utmz-600)*8;
   c = (utmx-410)*8;
   lseek(fdin,offset(r,c),0);
   for (i = 0; i< 80; i++){
      for(j = 0; j< 80; j++){

         read(fdin,&raw,2);
         veg = ((unsigned short)(raw & 0xe000) >> 13)+48;
         write(fdout,&veg,2);
         printf("%c",veg);
      }
      printf("\n");
   }
 close(fdout);
 close(fdin);
} /* main */

/* This calculates the startpoint for the gridsquare within the          */
/* 36 Km by 35 Km database.The 6400 represents the number of data points  */
/* per grid square. The 35 is the number of grid squares in the north/south */
/* direction.                                                             */

offset(r,c)
int r,c;
{
 return ( 2 * (
               (6400 * 35 * (int)(c/80))
             + (6400      * (int)(r/80))
             + (       80 * (c%80))
             + (            (r%80))
```

```
            )
        );
}

/*******End of file VDB2.C ***************************************/

File files.h
/* This file sets the paths required for input/output to vdb2.c     */

#define MASTER_DMA_DTED_FILE "/usr/work/cdec/DTED/terrain.dat"
#define OUTPUTFILE "usr/work/wade/thesis/rawdata.h"
#define PRINTFILE "usr/work/wade/thesis/rawdata.p"

/*******End of file files.h ***************************************/
```

APPENDIX C SOURCE CODE FOR THE SPLIT-AND-MERGE PHASE

```
module split-and-merge.

import(reg /4).

/*$eject*/
body.

dynamic(min_index/1).
dynamic(max_index/1).
dynamic(segment/4).
dynamic(newregions/3).
dynamic(outputlist/1).
dynamic(intermedlist/1).
dynamic(currentregion/1).
dynamic(flag/1).
dynamic(pairlistin/1).
dynamic(pairlistout1/1).
dynamic(pairlistout2/1).
dynamic(sproditem/0).
dynamic(ssumitem/0).
dynamic(incr_global/2).
dynamic(vprodout/1).
dynamic(vprodin1/1).
dynamic(vprodin2/1).
dynamic(sumuplist/1).
dynamic(sumupsum/1).

split_threshold(100e-2).
mergethreshold(100e-2).

go1 :-
    set_state(global_stack,30000),
    set_state(main_stack,10000),
    system(compress_stacks),
    system(garbage_collection),
    display_statistics,
    handle_adj_regs(reg,newregions),
    split_merge(reg,newregions),
    print_output,
    nl.
```

```
display_statistics :-
    nl, stars,
    write_tab(18), write("SYSTEM STATUS"),nl,
    state(cpu_time,X),
    write("cpu time = "), write(X),write(" msec"), nl,
    state(main_stack, [U,C]),
    write("main stack used = "), write(U),nl,
    state(global_stack, [UG,CG]),
    write("global stack used = "), write(UG),nl,
    state(statement_table,[U1,C1]),
    write("statement table used = "), write(U1), nl,
    stars, nl.

stars :- write("************************************************************"),
    nl.

print_output :-
    newregions(Regnumber,NewPointList,Value),
    write("The "),write(Regnumber),write(" Region is:  "),nl,
    write(NewPointList),nl,
    write("The Value is: "), write(Value),nl,nl,
    fail.
print_output.

handle_adj_regs(INNAME, OUTNAME) :-
    del_all_statements(OUTNAME/3),handle_adj1(INNAME,OUTNAME),!.

handle_adj1(INNAME,OUTNAME) :-
    get_a_region(INNAME,Reg1,PL1,V1,NP1),
    get_a_region(INNAME,Reg2,PL2,V2,NP2),
    Reg1 =/= Reg2,
    display_statistics,
    check_if_adj(PL1,PL2,AdjPts),!,
    display_statistics,
    handle_adj2(OUTNAME,Reg1,PL1,V1,NP1,Reg2,PL2,V2,NP2,AdjPts),
    fail.
handle_adj1(INNAME,OUTNAME).

check_if_adj(PL1,PL2,AdjPts) :-
    real_intersection(PL1,PL2,AdjPts),
    length(AdjPts,LA), write(LA),nl,LA > 0,
    !.

handle_adj2(OUTNAME,Reg1,PL1,V1,NP1,Reg2,PL2,V2,NP2,AdjPts) :-
    display_statistics,
    handle_adj3(Reg1,PL1,NP1,AdjPts,NewPL1),
```

```
    OP =.. [OUTNAME, Reg1, NewPL1, V1],
    assertz(OP),
    handle_adj3(Reg2,PL2,V2,NP2,AdjPts,NewPL2),
    OQ =.. [OUTNAME, Reg2, NewPL2, V2],
    assertz(OQ),
    fail.

handle_adj2(INNAME,OUTNAME) :-  !.

handle_adj3(Reg,PL,NP,AdjPts,NewPL) :-
write("inside adj3"),nl,
    set_global(currentregion,PL),
    set_global(outputlist,[]),
    set_global(intermedlist,[]),
    set_global(flag,1),
    find_sublist_indices(PL,N1,N2,AdjPts),
    get_sublist(PL,1,N1,FPL1),get_sublist(PL,N2,NP,BPL1),
    length(AdjPts,LA),
    asserta(segment(Reg,N1,N2,100)),
    split_into_segments,
    asserta(segment(Reg,1,N1,100)),
    asserta(segment(Reg,N2,NP,100)),
    split_into_segments,
    set_global(flag,1),set_global(min_index,1),set_global(max_index,NP),
    adj_merge(Reg,N1,N2),
    build_interned_list(Reg),
    intermedlist(NewIndexList), write(NewIndexList),nl,
    length(NewIndexList,NewNum), write("new list length is "),
    write(NewNum), nl, buildnewlist(NewIndexList,NewPL),
    write("The Final Point List is "), write(NewPL), nl,
    display_segments_asserted,!.

adj_merge(R,P1,P2) :-
    display_segments_asserted,nl,
    flag(1), set_global(flag,0),
    doall(adj_merge_segment(R,P1,P2)),
    adj_merge(R,P1,P2).
adj_merge(R,P1,P2).

adj_merge_segment(R,P1,P2) :-
    segment(R,A,B,FAB),segment(R,C,D,FCD), B = C, B =/= P1,
    B =/= P2, not segment(R,A,D,FAD),
    merge_segment2(R,A,B,C,D,FAB,FCD).

adj_merge_segment(R,P1,P2) :-
    segment(R,A,B,FAB),segment(R,C,D,FCD), min_index(MIN),
```

```
    max_index(MAX),C = MIN, B = MAX, B =/= P1, C =/= P1,
    B =/= P2, C =/= P2,
    merge_segment3(R,A,B,C,D,FAB,FCD).

split_merge(INNAME, OUTNAME) :-
    split_merge2(INNAME,OUTNAME).

split_merge2(INNAME, OUTNAME) :-
    get_a_region(INNAME,RegNumber,PointList,Value,NumPoints),
    split_merge3(RegNumber,PointList,NumPoints),
    set_global(flag,1), build_intermed_list(RegNumber),
    intermedlist(NewIndexList), write(NewIndexList),nl,
    length(NewIndexList,NewNum), write("new list length is "),write(NewNum),
    buildnewlist(NewIndexList,NewPointList),
    write("The Final Point List is  "),write(NewPointList),nl,
    OP =.. [OUTNAME,RegNumber,NewPointList,Value],
    assertz(OP),
    fail.

split_merge2(INNAME, OUTNAME) :- !.

get_a_region(INNAME,RegNumber,PointList,Value,NumPoints) :-
    not segment(RegNumber,PointList,Value,NumPoints),
    Q =.. [INNAME,RegNumber,PointList,Value,NumPoints], call(Q).


/* This is the main control structure for the split and merge algorithm.  */

split_merge3(R,PointList,NumPoints) :-
    set_global(currentregion,PointList),
    set_global(outputlist,[]),
    set_global(intermedlist,[]),
    set_global(flag,1),
    asserta(segment(R,1,NumPoints,100)),
    split_into_segments,
    display_statistics,
    set_global(flag,1),
    set_global(min_index,1),
    set_global(max_index,NumPoints),
    merge_segments(R),
    display_segments_asserted,!.

split_into_segments :-
    display_segments_asserted,nl,
    flag(1), set_global(flag,0), doall(split_into_segment),
    split_into_segments.
```

```
split_into_segments.

split_into_segment :-
   segment(R,N1,N2,F),split_into_segment2(R,N1,N2,F).

split_into_segment2(R,N1,N2,F) :-
   split_threshold(T),
   F > T, average(N1,N2,N3), retract(segment(R,N1,N2,F)), !,
   display_statistics,
   write("The first sublist  indices and fit are: "),
   write(N1),write(" "),write(N3),write(" "),nl,
   find_fit(R,N1,N3,F13), asserta(segment(R,N1,N3,F13)),
   write(F13),nl,
   display_statistics,
   write("The second sublist indices and fit are: "),
   write(N3),write(" "),write(N2),write(" "),
   find_fit(R,N3,N2,F32),asserta(segment(R,N3,N2,F32)),
   write(F32),nl,
   set_global(flag,1),!.

merge_segments(R) :-
   display_segments_asserted,nl,
   flag(1), set_global(flag,0),
   doall(merge_segment(R)),
   merge_segments(R).
merge_segments(R).

merge_segment(R) :-
   segment(R,N1,N2,F12),segment(R,N3,N4,F34), N2 = N3,
   not segment(R,N1,N4,F3),
   nl,write(" N1= "),write(N1),write(" N2 = "),write(N2),
   write(" N3= "),write(N3),write(" N4 = "),write(N4),nl,
   merge_segment2(R,N1,N2,N3,N4,F12,F34).
merge_segment(R) :-
   segment(R,N1,N2,F12),segment(R,N3,N4,F34),
   min_index(MIN),max_index(MAX),
   N3 = MIN,
   N2 = MAX,
   write(" N1= "),write(N1),write(" N2 = "),write(N2),
   write(" N3= "),write(N3),write(" N4 = "),write(N4),nl,
   merge_segment3(R,N1,N2,N3,N4,F12,F34).


merge_segment2(R,N1,N2,N3,N4,F12,F34) :-
   display_statistics,
```

```
    find_fit(R,N1,N4,F14),
    write("Fit is "),write(F14),nl,
    mergethreshold(T), F14 <= T,
    asserta(segment(R,N1,N4,F14)),
    retract(segment(R,N1,N2,F12)),
    retract(segment(R,N3,N4,F34)),
    set_global(flag,1), !.
merge_segment3(R,N1,N2,N3,N4,F12,F34) :-
    display_statistics,
    find_wrap_fit(R,N1,N4,F14),
    write("Fit is : "),write(F14),nl,
    mergethreshold(T), F14 <= T,
    asserta(segment(R,N1,N4,F14)),
    retract(segment(R,N1,N2,F12)),
    retract(segment(R,N3,N4,F34)),
    set_global(min_index,N4),
    set_global(max_index,N1),
    set_global(flag,1), !.


display_segments_asserted :-
    segment(A,B,C,D), write("Reg= "), write(A),
    write("  Indices = "), write(B),
    write("  "), write(C),
    write(" Fit = "), write(D),
    nl, fail.
display_segments_asserted :- nl.

average(N1,N2,N3) :-
    NP1 is N1 + 1, NP1 =/= N2,
    N3 is ((N2 - N1) div 2) + N1,!.

find_fit(R,A,B,Fit) :-
    currentregion(X),
    A < B,
    get_sublist(X,A,B,SubList),nl,
    endptlsline(SubList,[C1,C2,C3]),
    lslinefit(SubList,[C1,C2,C3],Fit).

find_wrap_fit(R,A,B,Fit) :-
    currentregion(X),
    A > B,
    get_rest_list(X,A,SubList1),
    get_sublist(X,1,B,SubList2),
    append(SubList1,SubList2,SubList),
    endptlsline(SubList,[C1,C2,C3]),
```

```
    lslinefit(SubList,[C1,C2,C3],Fit).

build_intermed_list(R) :-
    flag(1),set_global(flag,0),
    doall(get_segments(R)),
    intermedlist(IL),
    unduplicate(IL,NewIL), sort(NewIL,NNewIL),
    set_global(intermedlist,NNewIL).

build_intermed_list(R) :- !.

get_segments(R) :-
    segment(R,N1,N2,Fit),
    get_segments2(R,N1,N2,Fit).

get_segments2(R,N1,N2,Fit):-
    cons_global(intermedlist,N1),
    cons_global(intermedlist,N2),
    set_global(flag,1),!.


buildnewlist(NewIndexList,NewPointList) :-
    currentregion(PointList),
    set_global(outputlist,[]),
    buildnewlist2(PointList,NewIndexList),
    outputlist(RNewPointList), reverse(RNewPointList,NewPointList),!.

buildnewlist2(PointList,[]).
buildnewlist2(PointList,[N|BNewIndexList]) :-
    get_point(PointList,N,Point),
    cons_global(outputlist,Point),
    buildnewlist2(PointList,BNewIndexList).


get_point([A|BPointList],N,A) :-
    N = 1, !.

get_point([A|BPointList],N,Point) :-
    NC is N - 1,
    get_point(BPointList,NC,Point), !.

/* get_sublist is called when List, N1, and N2 are bound: yields SubList */
get_sublist(List,N1,N2,SubList) :-
    length(List,LL), LL >= 2,
    get_rest_list(List,N1,L1),(NC2 is N2 - N1 + 2),
    get_rest_list(L1,NC2 ,L2),
```

```
    append(SubList,L2,L1).

get_rest_list(RestList,N,RestList) :-
    N = 1, !.

get_rest_list([A|L],N,Restlist) :-
    NC is N -1,
    get_rest_list(L,NC,RestList),!.

/* find_sublist_indices is called when List and SubList are bound.        */
find_sublist_indices(List,N1,N2,[First|BSubList]) :-
    last(BSubList,Last),
    find_sublist_indices2(List,1,N3,First),
    find_sublist_indices2(List,1,N4,Last),
    order_indices(N3,N4,N1,N2), !.

find_sublist_indices2([[X1,Y1]|L],NC,NC,[X2,Y2]):- closer(X1,X2), closer(Y2,Y2),
    !.
find_sublist_indices2([A|L],NC,N,Point) :-
    NC2 is NC + 1,find_sublist_indices2(L,NC2,N,Point), !.

order_indices(N1,N2,N1,N2) :- N1 <= N2,!.
order_indices(N1,N2,N2,N1) :- !.

*doall(P) :- not alltried(P).

*alltried(P) :- call(P), fail.
```

module **linear least-squares**.

```
*split_pairs(L,L1,L2) :- set_global(pairlistin,L), set_global(pairlistout1,[]),
   set_global(pairlistout2,[]), iteratelist(split_pair(L),pairlistin),
   pairlistout1(RL1), pairlistout2(RL2), reverse(RL1,L1), reverse(RL2,L2), !.
*split_pair(L) :- pop_global(pairlistin,[X,Y]), cons_global(pairlistout1,X),
   cons_global(pairlistout2,Y).

*reverse_pairs([],[]).
*reverse_pairs([[A,B]|LL],[[B,A]|RLL]) :- reverse_pairs(LL,RLL).


/* Calculates the fit of a least-squares 2D line through a set of points.  */

*lsline(PL,[M,-1,B]) :-
   split_pairs(PL,XL,YL), vprod(XL,XL,XXL),
   vprod(YL,YL,YYL), vprod(XL,YL,XYL),sumup(XL,SXL),sumup(YL,SYL),
   sumup(XXL,SXXL), sumup(XYL,SXYL), length(XL,N),
   M is ((N*SXYL)-(SXL*SYL))/((N*SXXL)-(SXL*SXL)),B is (SYL-(M*SXL))/N, !.

endptlsline([[X1,Y1]|BPL],[C1,C2,C3]) :-
   last(BPL,[X2,Y2]),
   C1 is (Y1 - Y2),
   C2 is (X2 - X1), C3 is ((X1 * Y2) - (X2 * Y1)),!.

*lslinefit(PL,[C1,C2,C3],Fit) :-
   abs(C1,AC1),abs(C2,AC2), AC1 > AC2,!,
   reverse_pairs(PL,RPL),
   lslinefit(RPL,[C2,C1,C3],Fit).

*lslinefit(PL,[C1,C2,C3],Fit) :-
   M is (0-C1)/C2, B is (0-C3)/C2,
   split_pairs(PL,XL,YL), vprod(XL,XL,XXL),
   vprod(YL,YL,YYL), vprod(XL,YL,XYL), sumup(XL,SXL), sumup(YL,SYL),
   sumup(XXL,SXXL), sumup(XYL,SXYL), length(XL,N), sumup(YYL,SYYL),
   SqFit is (M*M*SXXL) + (2*M*B*SXL) + SYYL + (N*B*B) + (-2 *M*SXYL)
   + (-2 *B*SYL),
   xsqrt(SqFit,A),D1 is M*M+1, xsqrt(D1,D),
   Fit is A/D,!.


/* Iterative vector processing */
*sumup(L,N) :- set_global(sumuplist,L), set_global(sumupsum,0),
   iteratelist(sumupitem,sumuplist), sumupsum(N), !.
*sumupitem :- pop_global(sumuplist,X), sumupsum(N), retract(sumupsum(N)),
   NpX is N+X, asserta(sumupsum(NpX)).
```

```
*vprod(V1,V2,V) :- set_global(vprodout,[]), set_global(vprodin1,V1),
    set_global(vprodin2,V2), iteratelist(vproditem,vprodin1),
    vprodout(RV), reverse(RV,V), ! .

*vproditem :- pop_global(vprodin1,X), pop_global(vprodin2,Y), XY is X*Y,
    cons_global(vprodout,XY).
*vsum(V1,V2,V) :- set_global(vsumout,[]),
    set_global(vsumin1,V1), set_global(vsumin2,V2),
    iteratelist(vsumitem,vsumin1), vsumout(RV), reverse(RV,V), !.
*vsumitem :- pop_global(vsumin1,X), pop_global(vsumin2,Y), XY is X+Y,
    cons_global(vsumout,XY).
*vdiff(V1,V2,V) :- set_global(vdiffout,[]), set_global(vdiffin1,V1),
    set_global(vdiffin2,V2), iteratelist(vdiffitem,vdiffin1),
    vdiffout(RV), reverse(RV,V), !.
*vdiffitem :- pop_global(vdiffin1,X), pop_global(vdiffin2,Y), XY is X-Y,
    cons_global(vdiffout,XY).
*sprod(V1,K,V) :- set_global(sprodout,[]), set_global(sprodin,V1),
    iteratelist(sproditem(K),sprodin), sprodout(RV), reverse(RV,V), !.
*sproditem(K) :- pop_global(sprodin,X), NX is X*K, cons_global(sprodout,NX).
*ssum(V1,K,V) :- set_global(ssumout,[]), set_global(ssumin,V1),
    iteratelist(ssumitem(K),ssumin), ssumout(RV), reverse(RV,V), !.
*ssumitem(K) :- pop_global(ssumin,X), NX is X+K, cons_global(ssumout,NX).

/* Management of global variables as single-argument facts */
*set_global(Name,Value) :-  OLDP =.. [Name,Oldvalue], retract(OLDP),
    P =.. [Name, Value], asserta(P), !.

*set_global(Name,Value) :- P =.. [Name,Value], asserta(P), !.

*cons_global(Name,I) :- P=..[Name,X], call(P), retract(P), NP=..[Name,[I|X]],
    asserta(NP), !.
*pop_global(Name,Value) :- P=..[Name,[Value|L]], call(P), retract(P),
    NP=..[Name,L], asserta(NP), !.
*incr_global(Name) :- P=..[Name,X], call(P), retract(P), Xp1 is X+1,
    NP=..[Name,Xp1], asserta(NP), !.

/* Forward-execution iteration, terminates when a given list is empty */
*iteratelist(Ipred,Lname) :- repeat, iterate2(Ipred), P=..[Lname,[]],
  call(P), !.
*iterate2(Ipred) :- call(Ipred), !.
*iterate2(Ipred).
```

```
module lists.

/* Various list-processing predicates                                    */
/* First, the basics                                                     */

*last([X],X) .
*last([X|L],Y) :-
   last(L,Y) .

*append([],L,L) .
*append([X|L],L2,[X|L3]) :-
   append(L,L2,L3) .

*reverse(L,R) :-
   reverse2(L,[],R) .

*reverse2([],L,L) :-
   ! .
*reverse2([X|L],R,S) :-
   reverse2(L,[X|R],S) .

/* Predicates defined from others                                        */

*unduplicate([],[]) :-
   ! .
*unduplicate([X|L],L2) :-
   member(X,L), !, unduplicate(L,L2) .
*unduplicate([X|L],[X|L2]) :-
   unduplicate(L,L2) .

*intersection([],L,[]).
*intersection([X|L1],L2,[X|L3]) :-
   member(X,L2), !,intersection(L1,L2,L3).
*intersection([X|L1],L2,L3) :-
   intersection(L1,L2,L3).

real_intersection([],L,[]).
real_intersection([X|L1],L2,[X|L3]) :-
   real_pts_member(X,L2), !,real_intersection(L1,L2,L3).
real_intersection([X|L1],L2,L3) :-
   real_intersection(L1,L2,L3).

real_pts_member([X1,Y1],[[X2,Y2]|L]) :- X1 =:= X2, Y1 =:= Y2, !.

real_pts_member(X,[Y|L]) :-
   real_pts_member(X,L) .
```

```
*member(X,[X|L]) .
*member(X,[Y|L]) :-
   member(X,L) .

*singlemember(X,[X|L]) :-
   !.
*singlemember(X,[Y|L]) :-
   singlemember(X,L).

module math.

/* Mathematical Formulas not implemented in M-Prolog                    */
*xsqrt(0,0) :-
   !.
*xsqrt(X,Y) :-
   X<1, !, square_bisection(Y,X,X,1).
*xsqrt(1,1) :-
   !.
*xsqrt(X,Y) :-
   X>1, RX is 1/X, xsqrt(RX,RY), Y is 1/RY.

*square_bisection(X,Y,LO,HI) :-
   X is (LO+HI)/2, square(X,S), close(S,Y), !.
*square_bisection(X,Y,LO,HI) :-
   MIDPOINT is (LO+HI)/2, square(MIDPOINT,S), S<Y, !,
   square_bisection(X,Y,MIDPOINT,HI).
*square_bisection(X,Y,LO,HI) :-
   MIDPOINT is (LO+HI)/2, square(MIDPOINT,S), S>=Y, !,
   square_bisection(X,Y,LO,MIDPOINT).

*close(X,Y) :-
   D is X-Y, D > -1.0E-6, D < 1.0E-6.

*closer(X,Y) :-
   D is X-Y, D > -1.0E-3, D < 1.0E-3.

*square(X,Y) :-
   Y is X*X.

*expon(X,Y) :-
   expon2(X,1,1,1,Y).

*expon2(X,N,S,T,S) :-
   T<1.0E-6, !.
*expon2(X,N,S,T,Y) :-
```

```
    TP1 is X/N*T, SP1 is S+TP1, NP1 is N+1, expon2(X,NP1,SP1,TP1,Y).

abs(X,X) :- X >= 0,!.
abs(X,Y) :- Y is 0 - X,!.

endmod.  /* split_merge */
```

LIST OF REFERENCES

1. Ballard, Dana H. and Brown, Christopher M., *Computer Vision*, Prentice Hall, Inc., 1982.

2. Pavlidis, Theo, *Algorithms for Graphics and Image Processing*, Computer Science Press, Rockville, Maryland, 1982.

3. Roberts, L. G., "Machine Perception of three-dimensional solids," *Optical and Electro-optical Information Processing"*, MIT Press, 1965.

4. Ross, Ron S., *Planning Minimum-Energy Paths in an Off Road Environment with Anisotropic Traversal Costs and Motion Constraints*, Ph.D. Dissertation, Naval Postgraduate School, Monterey, California, June 1989.

5. Yee, Seung Hee, *Three Algorithms for Planar-Patch Terrain Modeling*, Masters Thesis, Department of Computer Science, Naval Postgraduate School, Monterey, California, June 1988.

6. Gonzalez, R.C., *Digital Image Processing*, Addison-Wesley Publishing Company, 1987.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center — 2
   Cameron Station
   Alexandria, Virginia 22304-6145

2. Library, Code 0142 — 2
   Naval Postgraduate School
   Monterey, California 93943-5002

3. Chief of Naval Operations — 1
   Director, Information Systems(OP-945)
   Navy Department
   Washington, D.C. 20350-2000

4. Department Chairman, Code 52 — 2
   Department of Computer Science
   Naval Postgraduate School
   Monterey, California 93943-5000

5. Curriculum Officer, Code 37 — 1
   Computer Technology
   Naval Postgraduate School
   Monterey, California 93943-5000

6. Professor Neil C. Rowe, Code 52Rp — 2
   Computer Science Department
   Naval Postgraduate School
   Monterey, California 93943-5000

7. Professor Sehung Kwak, Code 52Kw — 1
   Computer Science Department
   Naval Postgraduate School
   Monterey, California 93943-5000

8. CPT Roderick K. Wade — 2
   HQDA, Artificial Intelligence Center
   ATTN: CSDS-AI
   Washington, D.C. 20310-0200

Thesis
W15 Wade
c.1 A split-and-merge method for creating polygonal homogeneous-vegetation regions from digitized terrain data.